USENIX

conference

proceedings

HotOS duress
The summer sky is tempting
But I stayed inside.

A haiku-free zone?
No! HotOS thrives on prose
Spring forth and make art

Non-linear cliffs
Graceless degradation—bad
Avoid the fall, please

Enter session late?
Thou shalt not ask a question
Fall on your sword now

Virtual machines
are monolithic kernels
all seasons return

More security
through better access control
hope springs eternal

Declarativeness
It is seeing a new spring
Consistently hard

A haiku free zone
Automation sometimes bad
The answer falls out

Language as lifestyle
but give up eating red meat
And use my language

Fashion show software
In winter we run system
Faster than a pig

Got no performance?
You need a new systems conf
Or some Viagra

Seven syllables
Surrounded by two times five
specification

Armando Fox says
The Internet is for porn
We can't disagree

Another haiku
Some would prefer that it
stops
Perhaps sonnets?

HotOS Outrage
So many try for humor
Or is it wisdom?

Degeneration
Self-referential haiku
Far too much to drink

Eat your own dogfood
Expire software frequently
Summer of new code

A virtual mood
The candles are in my mind
My space or yours, dear?

Butler transactions
This is not so outrageous
Exotic hardware

Great new thought springs
forth
Porn access is currency
Make others do work

Sex, drugs, rock and roll
Technology drivers all
Cultural winter?

HotOS X

Tenth Workshop on Hot Topics in Operating Systems

Santa Fe, NM, USA

June 12–15, 2005

Sponsored by
The USENIX Association,
in cooperation with the
IEEE Technical Committee on Operating Systems (TCOS)

Declarativeness
System administration
Seeing a new spring

Kirk thinks in haiku
But he is not a poet
He is a real geek

What is an OS?
Today it's an app server
Winter of kernels

Transactions are good
Butler says, so we believe
Who will implement?

When I cross borders
I must savor the moment
No RFID

Content-free grammars
Computer science buzzwords
HotOS rejects

Random prose makes sense
More sense than serious work
Standards fall quickly

Video slidehow
Imcompatabilities abound
The spoken word rules

It's Mark V Chaney
Didn't Rob do that years back?
I believe we're drunk

Virtual machines
Winter of our discontent
What shock and awe!

The net is for porn
Armando tells us it's so
What will fallout be?

Promises broken
Threads intertwined fall poorly
Events catch me up!

Virtual machine
Is it a microkernel?
Lollies shall be thrown

Sew with threads, so what!
Run fast with events, so what!
Summer bad, so what!

Schizophrenia
You are not general enough
Three is much better

Sheeer paranoia
Make my house a computer
Summer of loving

An outrageous thought
Please change the laws of
physics
And the sky will fall

Systems research sucks
No relevance can be found
Ideas, spring forth!

Proceedings of HotOS X: Tenth Workshop on Hot Topics in Operating Systems Santa Fe, New Mexico, June 12–15, 2005

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: http://www.usenix.org

The price is $25.
Outside the U.S.A. and Canada, please add
$5 per copy for postage (via air printed matter).

ISBN 1-931971-36-6

USENIX Association

Proceedings of
HotOS X
Tenth Workshop in
Operating Systems

Sponsored by USENIX
in cooperation with
the IEEE Technical Committee on Operating Systems
(TCOS)

June 12–15, 2005
Santa Fe, NM, USA

Conference Organizers

Program Chair

Margo Seltzer, *Harvard University*

Program Committee

Hari Balakrishnan, *Massachusetts Institute of Technology*
Mary Baker, *Hewlett-Packard Labs*
Peter Drushel, *Rice University*
Stephanie Forrest, *University of New Mexico*
Armando Fox, *Stanford University*
Robert Grimm, *New York University*
Butler Lampson, *Microsoft Research*
Sharon Perl, *Google, Inc.*
John Reumann, *IBM Research*
Christopher Small, *Vanu*
Feng Zhao, *Microsoft Research, USA*

The USENIX Association Staff

HotOS X: Tenth Workshop on Hot Topics in Operating Systems

June 12–15, 2005

Santa Fe, NM, USA

Monday, June 13, 2005

Religious Wars

Session Chair: Robert Grimm, New York University

Storage

Session Chair: Butler Lampson, Microsoft Research

Outside the Comfort Zone

Session Chair: Sharon Perl, Google, Inc.

It's Not AI, It's Systems

Session Chair: Stephanie Forrest, University of New Mexico

Tuesday, June 14, 2005

Cleaning Up the Mess We've Made
Session Chair: Christopher Small, Vanu

Approaches to OS Research
Session Chair: Peter Druschel

Short Presentations

Distribution
Session Chair: Mary Baker, Hewlett-Packard Labs

Security
Session Chair: Armando Fox, Stanford University

Wednesday, June 15, 2005

Sensor Nets
Session Chair: Doug Terry, Microsoft Research

Breakout Session
Session Chair: Margo Seltzer, Harvard University

Index of Authors

Are Virtual Machine Monitors Microkernels Done Right?

Steven Hand, Andrew Warfield, Keir Fraser,
Evangelos Kotsovinos, Dan Magenheimer†
University of Cambridge Computer Laboratory
† HP Labs, Fort Collins, USA

1 Introduction

At the last HotOS, Mendel Rosenblum gave an 'outrageous' opinion that the academic obsession with microkernels during the past two decades produced many publications but little impact. He argued that virtual machine monitors (VMMs) had had considerably more practical uptake, despite—or perhaps due to—being principally developed by industry.

In this paper, we investigate this claim in light of our experiences in developing the Xen [1] virtual machine monitor. We argue that modern VMMs present a practical platform which allows the development and deployment of innovative systems research: in essence, VMMs are microkernels done right.

We first compare and contrast the architectural purity of microkernels with the pragmatic design of VMMs. In Section 3, we discuss several technical characteristics of microkernels that have proven, in our experience, to be incompatible with effective VMM design.

Rob Pike has irreverently suggested that "systems software research is irrelevant", implying that academic systems research has negligible impact outside the university. In Section 4, we claim that VMMs provide a platform on which innovative systems research ideas can be developed and deployed. We believe that providing a common framework for hosting novel systems will increase the penetration and relevance of systems research.

2 Motivation and μHistory

Microkernels and virtual machine monitors are both well explored areas of operating systems research dating back more than twenty years. Both areas have focused on a refactoring of systems into isolated components that communicate across well-defined, typically narrow interfaces. Despite considerable structural similarities, the two research areas are remarkable in their differences: Microkernels received considerable attention from academic researchers through the eighties and nineties, while VMM research has largely been the bailiwick of industrial research.

2.1 Microkernels: Noble Idealism

The most prolific academic microkernel ever developed was probably Mach [2]. A major research project at CMU, Mach's beginnings were in the Rochester Intelligent Gateway (RIG) [3] followed by the Accent kernel [4]. The key motivation to all of these systems was that the OS be "communication oriented"; that they have rigid, message-based interfaces between system components. Many of the abstractions used in Mach and later systems appeared initially in the RIG, including that of the *port*. However, the communications orientation of these systems originally intended to allow the distribution of system components across a set of dissimilar physical hosts.

The term "microkernel" was coined in response to the predominant monolithic kernels at the time. Microkernel advocates claimed that a smaller OS core would be easier to maintain, validate, and port to new architectures. A common theme throughout much of the microkernel work is that microkernels were *architecturally better* than monolithic kernels; from a research perspective they certainly are, as it is considerably easier to work on a single system component if that component is not entangled with other code.

Mach is hardly unique as an example of innovative microkernel projects. In the heyday of microkernels, many interesting systems were constructed including Chorus [5], Amoeba [6], and L3/L4 [7, 8]. Several of these evolved to show that microkernels, which were often criticized for poor performance, could match and even outperform commercial unix variants.

as one hour of maintenance for every one hundred hours of usage.[2] We claim that modern cars are considered dependable because they have an easily understood operation model consisting of regular fueling, regular oil changes, regular maintenance, and basically predictable, uninterrupted usage the rest of the time.

No open, general purpose software system can make a similar claim. They all must be patched frequently and regularly to fix flaws that open the system to malicious attack. They all can fail in ways that are inexplicable and unpredictable to ordinary users. Many of these users are afraid to change their system in even the slightest way, for fear of breaking them.

2.2 Security

Contemporary OS security systems were designed to protect users of a system against each other and to protect the OS from errant programs. These security architectures were developed in the quaint past when code came from trusted sources and networks mostly connected us with our friends and colleagues. In today's connected world, users and computers are surrounded by unscrupulous advertisers, petty criminals, and increasingly organized crime. In this world in which executable code can and does come from anywhere, the OS needs to protect user and system resources from potentially hostile code that a user runs either intentionally or unintentionally. This is a very hard problem given that desired code may do useful work!

To bring code into an OS security model, there must be a basic OS abstraction that represents the identity of code. The abstraction should also capture the provenance of the code as well as provide a means for checking code integrity. Once code is identifiable, we can imagine enforcing security policy pertaining to it.

Code identity alone, however, is not sufficient. Software components interact in exceedingly complex ways, and many such interactions are security-relevant. We can expect the next generation of attacks to exploit unplanned and unprotected interactions between software components. There is fertile ground for research in understanding how to prevent such attacks by design.

The Java [12] and Common Language Infrastructure (CLI)[3] [24] programming environments have explored some of these issues. However, the security models in these systems are complex and largely separate from OS models.

[2] An oil change (1 hour) every 5,000 miles (100 hours at 50 miles/hour) is typical and does not take into account other preventive maintenance, which typically takes a car out of commission for an entire day.

[3] Microsoft's commercial implementation of the CLI is known as the Common Language Runtime (CLR). The CLR is the core of Microsoft's .NET Framework.

2.3 System Configuration

Contemporary operating systems contain abstractions for many components of modern applications, such as processes, threads, and shared libraries, but applications and their dependencies are only informally characterized. Lacking a strong concept of an application's complete configuration, the OS has no mechanisms to guarantee the integrity or provenance of an application. A system is only as stable as its most fragile component, which cannot be identified in current systems; systems which provide no easy way to distinguish application components intermixed in file systems and configuration registries.

Consider, for example, the case of applications colliding in their usage of shared spaces such as file systems or configuration registries. The installation of one application may corrupt or irreversibly alter the configuration of another via changes to a file or registry. The "DLL Hell" problem in Windows systems occurs when one application overwrites a common shared library with a version incompatible with an existing application. Similar problems can occur when an application overwrites configuration information mapping from document extensions to applications. To compensate for the absence of OS managed applications, users resort to ad-hoc application isolation techniques, such as jails [14] or virtual machine monitors, such as VMware [9] and Xen [3].

2.4 System Extension

Since no monolithic system can satisfy all users, most complex software lets users load code to extend functionality. Dynamically loaded extensions are found as widely as device drivers in kernels and spelling checkers in word processors. Whether in the OS or an application, most extensions are loaded directly into a host address space with no hard interface, protection boundary, or clear distinction between host and extension code. Extension through in-process code loading appears flexible and attractive, but due to a lack of isolation, extensions are a major source of software reliability and security problems. For example, faulty device drivers cause a large fraction of Windows and Linux failures [22].

A number of OS research efforts, including Exokernel [13], SPIN [5], VINO [21], and Nooks [22] have sought safer OS extension without addressing the more general problem of application extension. Pragmatically, each of these systems provided domain-specific models for OS extensions. Software fault isolation (SFI) [23], one of the few research efforts to consider application extension, limits an extension to a subset of an application's address space. However, the overhead for SFI is quite high and still exposes published data structures to corruption by the extension.

isolated and cannot degrade the stability of the system as a whole.

Consider, as a counterexample to external paging, the storage virtual machine used in Parallax [17]. In this case a storage VM is used to serve block storage to a collection of client VMs. A crash in the storage server could compromise the function of its clients, but not of the system as a whole: in particular, Xen itself does not depend on the correctness of the storage VM to function. Moreover, the dependency between the storage VM and its clients is *explicit*: the isolation between dependent VMs can be increased by separating the storage VM into multiple instances. This is essentially just the traditional trade-off between isolation and sharing which is observed in the design of any system.

3.2 Make IPC Performance Irrelevant

IPC performance is arguably the most revered hallmark of microkernel research. As message-based communication between system components is crucial to the operation of any microkernel, the literature is saturated with papers measuring IPC performance, improving IPC performance, and even questioning the relevance of IPC performance. However in our experience fast IPC is not a critical design concern in the construction of high-performance VMMs.

There are a number of reasons why we can avoid relying on fast, typically synchronous, IPC mechanisms. First, since VMMs hold isolation to be a key goal, IPC between virtual machines is considerably less common in general. This is a natural consequence of the fact that VMM design considers entire operating systems to be the unit of scheduling and protection: hence synchronization and protected control transfer are only necessary when two virtual machines wish to explicitly communicate.

Secondly, we have determined that a clear separation between *control* and *data* path operations allows us to optimize for the common case. In particular, we observe that by explicitly setting up communication channels, we can perform potentially expensive permission and safety checks at initialization time and then elide validation during more frequent data path operations. This decoupling furthermore allows higher-level communcation mechanisms great freedom in how they are implemented.

A particular example of this is seen in the implementation of *control*- and *device-channels* within Xen. Both of these are built upon a simple asynchronous unidirectional event mechanism which is the only communications primitive provided by Xen. However by combining pairs of events with shared memory, we can build both synchronous IPC for control operations and asynchronous producer-consumer rings for bulk, batched, data transfer. Even these latter allow considerable flexibility in use: by determining how often notifications are generated or waited upon, one can explicitly trade-off throughput and latency.

The difference between approaches to communication between isolated components is a very interesting example of the idealism versus pragmatism dichotomy described in the previous section. Microkernel designers view systems as sets of components that interact over IPC-, and potentially RPC-, based interfaces: they consider these interactions as procedure calls, in which the entire system is a collection of well-isolated components. VMM designers do not assume anywhere near the same degree of coherency within their systems: where VMs do communicate, they may not only be written in separate programming languages, but may also be running completely different operating systems. A consequence of this is that communications within VMMs typically looks like interactions with devices: a simple asynchronous control path combined with fixed-format transparent bulk data transfer.

3.3 Treat the OS as a Component

The final important difference between VMMs and microkernels is that of the *granularity of componentization*. By positioning themselves as a response to monolithic kernels, microkernels focused on dividing the functional units of an OS into discrete parts. A practical problem faced by microkernel developers is that which faces any new OS effort: by changing the API visible to applications, an OS forfeits the complete set of software available to existing systems. As such, most microkernel projects were left spending considerable effort to implement emulation interface layers for existing OSes.

VMMs differ significantly here in that their *a priori* intention is to support existing operating systems. For example, out-of-the-box code, compiled to be executable on a range of existing OSes, can be run on a guest operating system on top of Xen. This reduces the cost of entry for users and applications, makes virtualization attractive and practical for a wider community, and addresses two of the main problems of microkernel systems — the difficulty in attracting a substantial user base, and the challenge in keeping microkernel operating systems up to date with the feature sets of existing OSes.

By supporting existing OSes, VMMs need only justify the potential performance overheads they incur in order to be an attractive option. As shown in [1] and independently verified in [18], the overhead imposed by Xen is very small.

Secondly, VMMs appeal to developers because they present a *familiar development environment*. Using existing OSes as fundamental blocks of componentization allows developers to continue using the same tool set that they have on their existing system, freeing them to concentrate on more important issues.

The Parallax storage system [17], mentioned earlier, is an example of the sort of componentization that VMMs allow: The storage VM is a set of daemons running on Linux in an isolated virtual machine. The system can be used by any OS that runs on Xen because it provides the same block interface that Xen's existing block virtualization uses. Parallax provides an extension to an OS function, an ability touted by microkernels, but does it in a familiar development environment, using existing OS drivers, and providing support in turn for a range of client OSes. Moreover, the implementation is independent from both Xen and client OS code: provided that the block interface remains common, the OS extension itself does not depend on the source of the client OSes or the VMM.

Similar benefits accrue for the developers of the VMM itself: for example, Xen makes extensive use of existing tools for network routing, disk management, and configuration as part of the control software running in the privileged management VM.

The size of components — i.e., guest OSes — running on a VMM can be adjusted, depending on the functionality required from them. One example is *ttylinux*, a minimalistic Linux distribution, providing multi-tasking, multi-user, and networking capabilities within less than 4 megabytes of operating system size. It is also easy to build a simple single-threaded 'library OS' which enables the use of extremely lightweight components when desired for security or performance reasons.

4 The future for VMMs

Having illustrated what we feel are the key differences between microkernel and VMM design, we now consider how VMMs may be used to realize many of the research benefits achieved by the microkernel community. These include narrow interfaces between system components providing easy **extensibility** of device and OS functionality, a small code base that can guarantee **security** more easily than monolithic kernels, and strong isolation providing opportunities for improved **manageability**.

Narrow interfaces between system components are crucial in facilitating extensibility. The clean IPC interfaces provided by microkernels allowed researchers the ability to focus on specific system components without becoming entangled in unrelated code. Similarly, the narrow interfaces present in Xen allow devices and OSes to be easily extended. Xen's device architecture has allowed device drivers to be isolated in a separate VM for dependability [19], and permitted low-level interfaces to be extended without necessitating modification of the target OS or VMM [20]. Indeed, it seems very likely that the exploration of how services and management will be structured in a multi-OS VMM system will continue to present many exciting research opportunities.

A further advantage of narrow interfaces, coupled with a minimal privileged kernel, is the tractability of achieving a high degree of confidence in the security of a system. This has been explored in the microkernel community by projects such as Flask [21] and EROS [22]. Several groups have expressed interest in developing these ideas for Xen, using concepts from projects such as the Flask-derived SELinux.

A final avenue of innovation realized recently by VMMs has been to explore less performance-centric aspects of systems development. As with the examples above, VMMs are a promising platform because these so-called 'ilities' can be developed and applied to existing systems. For example, live OS migration [23] allows a running OS to be relocated to a new physical host, empowering administrators to better manage physical resources. The ability to 'rewind' a VM's state has been used for intrusion detection [24], debugging [25] and administration [26].

5 Conclusion

Despite having dissimilar motivations and origins, microkernels and VMMs share many architectural commonalities. In this paper we have attempted to illustrate some of the technical separations between the two classes of system that, in our opinion, have favoured the success of VMMs in recent years. More importantly though, we posit that—despite the decline in microkernel research— modern VMMs, Xen in particular, are in fact a specific point in the microkernel design space; that VMMs are microkernels done right. In light of this opinion, we observe that many of the advantages realized through the structure of microkernel systems may be similarly developed above a VMM. Moreover, because VMMs run commodity operating systems and applications we claim that they present a valuable platform for innovative systems research to have impact outside the academic laboratory.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[2] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proc. Summer USENIX Conference*, June 1986.

[3] E. Ball, J. Feldman, J. Low, R. Rashid, and P. Rovner. RIG, Rochester's Intelligent Gateway: System overview. In *Proc. 2nd International Conference on Software Engineering*, page 132, 1976.

[4] R. Rashid and G. Robertson. Accent: A communication oriented network operating system kernel. In *Proc. 8th ACM Symposium on Operating Systems Principles (SOSP)*, pages 64–75, 1981.

[5] V. Abrossimov, M. Rozier, and M. Gien. Virtual memory management in chorus. In *Proc. European Workshop on Process in Distributed Operating Systems and Distributed Systems Management*, pages 45–59, 1990.

[6] S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, 1990.

[7] J. Liedtke. Improving IPC by kernel design. In *Proc. 14th ACM Symposium on Operating Systems Principles (SOSP)*, December 1993.

[8] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The Performance of μ-Kernel-Based Systems. In *Proc. 16th ACM Symposium on Operating Systems Principles (SOSP)*, October 1997.

[9] R. Adair, R. Bayles, L. Comeau, and R. Creasy. A virtual machine system for the 360/40. Technical Report 320-2007, IBM Corporation, Cambridge Scientific Center, May 1966.

[10] R. Goldberg. Architectural principles for virtual computer systems. PhD thesis, Harvard University, 1972.

[11] A. Whitaker, M. Shaw, and S. Gribble. Scale and performance in the Denali isolation kernel. In *Proc. 5th Symposium on Operating System Design and Implementation (OSDI)*, December 2002.

[12] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proc. 2nd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–151, October 1996.

[13] J. Dike. User-mode Linux. In *Proc. 5th Annual Linux Showcase and Conference*, November 2001.

[14] D. Engler, F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP)*, December 1995.

[15] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbaims, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. 14(7):1280–1297, September 1996.

[16] M. Young, A. Tevanian, R. F. Rashid, D. B. Golub, J. L. Eppinger, J. Chew, W. J. Bolosky, D. L. Black, and R. V. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proc. 11th ACM Symposium on Operating Systems Principles (SOSP)*, 1987.

[17] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand. Parallax: Managing storage for a million machines. In *Proc. 10th Workshop on Hot Topics in Operating Systems (HotOS X*.

[18] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J. Matthews. Xen and the art of repeated research. In *Proc. USENIX Annual Technical Conference*, June 2004.

[19] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proc. ACM OASIS Workshop*, 2004.

[20] A. Warfield, K. Fraser, S. Hand, and T. Deegan. Facilitating the development of soft devices. In *Proc. USENIX Annual Technical Conference*, April 2005.

[21] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *Proc. Eighth USENIX Security Symposium*, August 1999.

[22] J. Shapiro, J. Smith, and D.Farber. EROS: a fast capability system. In *Symposium on Operating Systems Principles*, 1999.

[23] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[24] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.

[25] S. King, G. Dunlap, and P. Chen. Debugging operating systems with time-traveling virtual machines. In *Proc. USENIX Annual Technical Conference*, 2005.

[26] A. Whitaker, R. Cox, and S. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *Proc. 6th Symposium on Operating System Design and Implementation (OSDI)*, pages 77–90, December 2004.

OS Verification — Now!

Harvey Tuch Gerwin Klein Gernot Heiser
National ICT Australia
University of New South Wales
(harvey.tuch|gerwin.klein|gernot)@nicta.com.au

Abstract

Hard, machine-supported formal verification of software is at a turning point. Recent years have seen theorem proving tools maturing with a number of successful, real-life applications. At the same time, small high-performance OS kernels, which can drastically reduce the size of the trusted computing base, have become more popular. We argue that the combination of those two trends makes it feasible, and desirable, to formally verify production-quality operating systems — now.

1 Introduction

There is increasing pressure on providing a high degree of assurance of a computer system's security and functionality. This pressure stems from the deployment of computer systems in life- and mission-critical scenarios, and the need to protect computing and communication infrastructure against attack. This calls for end-to-end guarantees of systems functionality, from applications down to hardware.

While security certification is increasingly required at higher system levels, the operating system is generally trusted to be secure. This clearly presents a weak link in the armour, given the size and complexity of modern operating systems.

However, there is a renewed tendency towards smaller operating system kernels which could help here. This is mainly motivated by two increasingly popular scenarios:

Trusted applications and legacy software The general trend towards standard APIs and COTS technology (e.g. Linux) is even reaching safety- and security-critical embedded systems. Similarly, emerging applications on personal computers and home/mobile electronics require digital rights management and strong protection of cryptographic keys in electronic commerce. In both cases it is necessary to run large legacy systems alongside highly critical components to provide desired functionality, without the former being able to interfere with the latter. This requirement is met by de-privileging the legacy system and using a small kernel or monitor to securely switch between the trusted and untrusted subsystems, as in L4Linux, and processor manufacturers are moving towards hardware support for such partitioning (ARM TrustZone and Intel LaGrande).

Secure and efficient multiplexing of hardware This scenario partitions a system into isolated, de-privileged peer subsystems, typically several copies of the same or different full-blown operating systems. The partitioning may be based on full virtualisation (as in VMWare), or para-virtualisation, as in Xen and Denali. The underlying privileged *virtual-machine monitor* or *hypervisor* is typically of much smaller size than the operating systems running in individual partitions.

Both scenarios require an abstraction layer of software far smaller than a traditional monolithic OS kernel. For the rest of this paper we refer to this layer simply as the *kernel*, since the distinction between hypervisor, microkernel and protection-domain management software is not of relevance here.

The reduction in size, compared to traditional approaches, already goes a long way towards making the kernel more trustworthy. Standard methods for establishing the trustworthiness of software, such as testing and code review (while they inherently cannot guarantee absence of faults) work better on a smaller code base.

Recently, algorithmic techniques, like static analysis and model checking, have achieved impressive results in bug hunting in kernel software [8]. However, they cannot provide confidence in full functional correctness, nor can they give hard security guarantees.

The only real solution to establishing trustworthiness is formal verification, *proving* the implementation correct. This has, until recently, been considered an intractable proposition — the OS layer was too large and complex for poorly scaling formal methods. In this paper we argue that, owing to the combination of improvements in formal methods and the trend towards smaller kernels, full formal verification of real-life kernels is now within reach.

In the next section we give an overview of formal verification and its application to kernels. In Section 3 we examine the challenges encountered and experience gained in a pilot project that successfully applied formal verification to the L4 microkernel utilising the Isabelle theorem prover.

2 Formal verification

Formal verification is about producing strict mathematical proofs of the correctness of a system. But what does this mean? From the formal-methods point of view, it means that a formal *model* of the *system* behaves in a manner that is consistent with a formal *specification* of the *requirements*. This leaves a significant semantic gap between the formal verification and the user's view of correctness [6]. The user (e.g. application programmer) views the system as "correct" if the behaviour of its object code on the target hardware is consistent with the user's interpretation of the (usually informally specified) API. Bridging this semantic gap is called *formalisation*. This is shown schematically in Fig. 1.

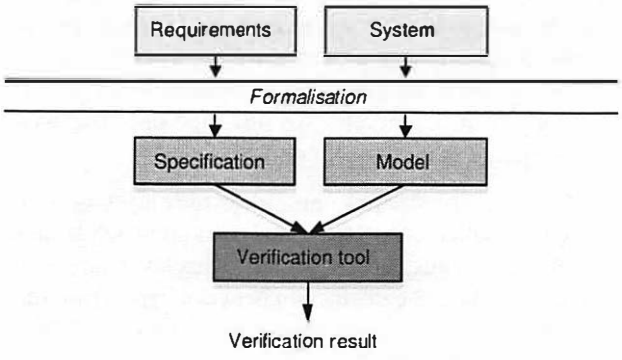


Figure 1: Formal verification process

Verification technology At present there are two main verification techniques in use: model checking and theorem proving. *Model checking* works on a model of the system that is typically reduced to what is relevant to the specific properties of interest. The model checker then exhaustively explores the model's reachable state space to determine whether the properties hold. This approach is only feasible for systems with a moderately-sized state space, which implies dramatic simplification. As a consequence, model checking is unsuitable for establishing a kernel's full compliance with its API. Instead it is typically used to establish very specific safety or liveness properties. Furthermore, the formalisation step from system to model is quite large, commonly done manually and therefore error prone. Hence, model checking usually does not give guarantees about the actual system. Model checking has been applied to the OS layer [17] and has shown utility here as a means of bug discovery in code involving concurrency. However, claims of implementation verification are disputable due to the manual abstraction step. Tools like SLAM [2] can operate directly on the kernel source code and automatically find safe approximations of system behaviour. However, they can only verify relatively simple properties, such as the correct sequencing of operations on a mutex — necessary but not sufficient for correct system behaviour.

The *theorem proving* approach involves describing the intended properties of the system and its model in a formal logic, and then deriving a mathematical proof showing that the model satisfies these properties. The size of the state space is not a problem, as mathematical proofs can deal with large or even infinite state spaces. This makes theorem proving applicable to more complex models and full functional correctness.

Contrary to model checking, theorem proving is usually not an automatic procedure, but requires human interaction. While modern theorem provers remove some of the tedium from the proof process by providing rewriting, decision procedures, automated search tactics, etc, it is ultimately the user who guides the proof, provides the structure, or comes up with a suitably strong induction statement. While this is often seen as a drawback of theorem proving, we consider it its greatest strength: It ensures that verification does not only tell you *that* a system is correct, but also *why* it is correct. Proofs are developed interactively with this technique but can be checked automatically for validity once derived, making the size and complexity of the proof irrelevant to soundness.

Verifying kernels What do the models and specifications look like in kernel verification?

Clearly a kernel needs to implement its API, so the specification is typically a formalisation of this API. This is created by a manual process with a potential for misstatement, as APIs tend to be specified informally or at best semi-formally using natural languages, and are typically incomplete and sometimes inconsistent. It is then desirable to utilise a formalism such that the correspondence between the informal and formal specification is relatively easy to see even for OS developers who are no experts in formal methods.

The kernel model is ideally the kernel executing on the hardware. In reality it is preferable to take advantage of the abstraction provided by the programming language in which the kernel is implemented, so the model becomes the kernel's source-level implementation. This introduces a reliance on the correctness of the compiler and linker (in addition to the hardware, boot-loader and firmware).

Some criticisms are commonly voiced when considering OS verification. *Is there any point if we have to rely on compiler and hardware correctness?* With source-level verification, compiler and hardware correctness have become orthogonal issues — when we have the required formal semantics for the language and hardware, verification of these system components can be attempted independently to that of the OS. Both hardware

and compiler verification are currently active areas of research. It should be noted that the gap between formal model and implementation will always exist, even in the presence of a verified processor, since real hardware is a physical realisation of some model and its correct operation is beyond the scope of formal verification [6] — one cannot prove the absence of manufacturing defects for example. The aim of OS verification is to significantly reduce the larger gap between user requirements and implementation and hence gain increased confidence in system correctness. *Even if the kernel is verified, what has been gained when user-level applications such as file-systems are not?* In the first scenario described in the introduction, the question is really that of what do we need to verify to be able to claim the *trusted* applications are correct. The kernel provides the basic abstraction over the underlying hardware necessary to enforce the boundary between trusted and untrusted applications and allows the behaviour of untrusted applications to be abstracted away or ignored when verifying the trusted code. Trusted applications may also have some redeeming characteristics when it comes to verification — they should be relatively small in a well-designed TCB and may take advantage of higher-level languages. For a hypervisor no additional work remains after OS verification — if the correct resource management and isolation is provided at the OS level then there is no possibility of faulty or malicious code executing in one partition from influencing or knowing about another.

Proof-based OS verification has been tried in the past [13, 20]. The rudimentary tools available at the time meant that the proofs had to end at the design level; full implementation verification was not feasible. The verification of Kit [4] down to object code demonstrated the feasibility of this approach to kernel verification, although on a system that is far simpler than any real-life OS kernel in use in secure systems today. There is little published work from the past 10–15 years on this topic, and we believe it is time to reconsider this approach.

3 Challenges and Experiences

Since the early attempts at kernel verification there have been dramatic improvements in the power of available theorem proving tools. Proof assistants like ACL2, Coq, PVS, HOL and Isabelle have been used in a number of successful verifications, ranging from mathematics and logics to microprocessors [5], compilers [3], and full programming platforms like JavaCard [18].

This has led to a significant reduction in the cost of formal verification, and a lowering of the feasibility threshold. At the same time the potential benefits have increased, given e.g. the increased deployment of embedded systems in life- or mission-critical situations, and the huge stakes created by the need to protect IP rights valued in the billions.

Consequently, we feel that the time is right to tackle, once again, the formal verification of OS kernels. We therefore decided about a year ago to attempt a verification of a real kernel. We are among several current efforts with this goal, notably VFiasco [9], VeriSoft [19] and Coyotos [15]. We target the L4 microkernel in our work as it is one of the smallest and best performing general-purpose kernels around, is deployed industrially and its design and implementation is well understood in our lab.

As this is clearly a high-risk project, we first embarked on a pilot project in the form of a constructive feasibility study. Its aim was three-fold: (i) to formalise the L4 API, (ii) to gain experience by going though a full verification cycle of a (small) portion of actual kernel code, and (iii) to develop a project plan for a verification of the full kernel. An informal aim was to explore and bridge the culture gap between kernel hackers and theorists, groups which have been known to eye each other with significant suspicion.

The formalisation of the API was performed using the B Method [1], as there existed a significant amount of experience with this approach among our student population. While L4 has an unusually detailed and very mature (informal) specification of its API [12], it came as no surprise to us to find that it was incomplete and ambiguous in many places, and inconsistent in a few. Furthermore it was sometimes necessary to extract the intended and expected kernel behaviour from the designers themselves and, occasionally, the source code.

In spite of those challenges, this part of the project turned out not overly difficult, and was done by a final-year undergraduate student. The result was a formal API specification that is mostly complete, describing the architecture-independent system calls in the IPC and threads subsystem of L4. Non-determinism was used in places where the current API was not clear on specific behaviour, and optimisations present at the API level that contributed significant complexity yet only provided disputable performance gains were omitted. The remaining subsystem (virtual memory) was formalised separately in the verification part of the project described below. The B specification consists of about 2000 lines of code.

The full verification was performed on the most complex subsystem, the one dealing with mapping of pages between address spaces and the revocation of such mappings, corresponding to approximately 5% of the kernel source code. We formalised a significant part of this API section and derived a verified implementation based on the existing implementation but with a subset of its functionality. Its implementation consists of the page tables, the *mapping database* (used to keep track of mappings for revocation purposes), and the code for lookup and

manipulation of those data structures.

Since our view of the system is that of execution on an abstract machine corresponding to the implementation language, the lowest-level model must rely on the formal semantics of the source code language and hardware. The L4 kernel is written in a mixture of a (mostly-C-like) subset of C++ with some assembler code. While the complete formal semantics of systems languages is an active area of research [9, 14], a complete semantics is not required. For our purpose it sufficed to have a semantics for the language subset actually used in the verified code. The code derived during this work was based on the data structures and algorithms in the existing implementation, but we had the freedom to make changes to remain in a safe subset of C++. Such changes are acceptable as long as they have no significant performance impact.

Semantics for the assembler code could be derived from the hardware model. This was not tackled in our pilot project, as the slice was implemented without resorting to assembler (our work is based on ARM processors, which feature hardware-loaded TLBs). We did, however, formalise some aspects of the hardware, such as the format of page table entries. In principle, processor manufacturers could provide their descriptions of the ISA level in a HDL to facilitate this, in practise this rarely happens. Instead one typically uses ISA reference manuals as a basis for formalisation. Hardware models of commercial microprocessors such as x86 and ARM [7] are available. While these are presently somewhat incomplete for kernel verification purposes, they should be extendable without major problems.

We use higher-order logic (HOL) as our language for system modelling, specification and refinement, specifically the instantiation of HOL in the theorem prover Isabelle. HOL is an expressive logic with standard mathematical notation. Terms in the logic are typed, and HOL can directly be used as simple functional programming language. HOL is consequently unesoteric for programmers with a computer science background. We are using this functional language to describe the behaviour of the kernel at an abstract level. This description is then *refined* inside the prover into a program written in a standard, imperative, C-like language. In a refinement some part the state space is made more concrete, substitutions for operations for the new state space are described and proven to simulate the abstract operations. For example, an abstract albeit simplistic view of the page table for an address space is a function mapping virtual pages to page table entries. Refinement of this would replace the function with a page table data structure such as a multi-level page table and the corresponding insertion and lookup procedures.

The abstract description is at the level of a reference manual and relatively easy to understand. This is the level we use for analysing the behaviour of the system and for proving additional simple safety properties, such as the requirement that the same virtual address can never be translated to two different physical addresses. The abstract model is operational, essentially a state machine. This is close to the intuition that systems implementors have of kernel behaviour as an extended hardware machine, and has an associated well-understood hierarchical refinement methodology. An operational model for kernel behaviour in HOL then helps minimising the gap between requirements and specification. At the end of the refinement process stands a formally verified imperative program. A purely syntactic translation then transforms this program into ANSI C. A detailed description of this process can be found elsewhere [11, 16].

We found Isabelle suitable for the task. It is mature enough for use in large-scale projects and well-documented, with a reasonably easy-to-use interface. Being actively developed as an open source tool, we are able to extend it and (working with the developers) to fix problems should they arise.

During the process of formalising the VM subsystem we discovered several places in the existing semi-formal description and reference manual where significant ambiguities existed, and some inconsistencies with implementation behaviour. The ordering of internal operations in the system calls responsible for establishing and revoking VM mappings, *map*, *grant* and *unmap*, was underspecified, leading to problems when describing a formal semantics. A potential security problem could result from one of the inconsistencies found.

An interesting experience was that the expected culture clash between kernel hackers and formal methods people was a non-issue. The first author of this paper is a junior PhD student with significant kernel design and implementation experience. He obtained the necessary formal methods background within two months to the degree where he could productively perform proofs in Isabelle. It took about the same time for all participants to gain an appreciation of the other side's challenges. This is one of the reasons that we believe that the full verification of L4 is achievable.

However, we are convinced that some important requirements must be met for such a project to have a chance. It is essential that some of the participants have significant experience with formal methods and a good understanding of what is feasible and what is not, and how best to approach it. On the other hand, it is essential that some of the participants have a good understanding of the kernel's design and implementation, the trade-offs underlying various design decisions, and the factors that determine the kernel's performance. It must

be possible to change the implementation if needed, and that requires a good understanding of changes that can be done without undermining performance.

4 Looking Ahead

The challenges for formal verification at the kernel level relate to performance, size, and the level of abstraction. Runtime performance of the verified code is one of the highest priorities in operating systems, particularly in the case of a microkernel or virtual-machine monitor, which is invoked frequently. Software verification has traditionally not focused on this aspect — getting it verified was hard enough. Size is a limiting factor as well. Even a small microkernel like L4 measures about 10,000 lines of code. Larger systems have been verified before, but only on an abstract description, not on the implementation level. Compared to application code, the level of abstraction is lower for kernel code. Features like direct hardware access, pointer arithmetic and embedded assembly code are not usually the subject of mainstream verification research.

Another practically important issue is ensuring that verified code remains maintainable. In principle, every change to the implementation might invalidate the verification. The extent to which this occurs will depend on the nature of the change. Hand optimisation of the IPC path, for example, may require less work to reestablish correctness than changes to system call semantics, since in the optimisation case higher-level abstraction proofs remain valid. The fact that the proofs are machine-checked makes it easy to determine which proofs are broken by the change, and techniques such as a careful, layered proof structure and improved automation for simple changes help to make this problem easier to handle. Whether this is enough remains an open question.

We believe that full formal specification of the kernel API prior to kernel implementation is desirable. The benefits of having a complete, consistent and unambiguous reference for kernel implementors, users and verifiers is clear, and the effort required is modest when compared with either implementation or verification.

The investment for the virtual memory part of the pilot project was about 1.5 person years. All specifications and proofs together run to about 14,000 lines of proof scripts. This is significantly more than the effort invested in the virtual memory subsystem in the first place, but it includes exploration of alternatives, determining the right methodology, formalising and proving correct a general refinement technique, as well as documentation and publications.

We estimate that the full verification of L4 will take about 20 person years, including verification tool development. This sounds a lot, but must be seen in relation to the cost of developing the kernel in the first place, and the potential benefits of verification. The present kernel [12] was written by a three-person team over a period of 8–12 months, with significant improvements since. Furthermore, for most of the developers it was the third in a series of similar kernels they had written, which meant that when starting they had a considerable amount of experience. A realistic estimate of the cost of developing a high-performance implementation of L4 is probably at least 5–10 person years.

Under those circumstances, the full verification no longer seems prohibitive, and we argue that it is, in fact, highly desirable. The kernel is the lowest and most critical part of any software stack, and any assurances on system behaviour are built on sand as long as the kernel is not shown to behave as expected. Furthermore, formal verification puts pressure on kernel designers to simplify their systems, which has obvious benefits for maintainability and robustness even when not yet formally verified.

There is a saying that the problem with engineers is that they cheat in order to get results, the problem with mathematicians is that they work on toy problems in order to get results, and the problem with program verifiers is that they cheat on toy problems in order to get results. We are ready to tackle the real problem without cheating.

Acknowledgements

We would like to thank all those who contributed to the part of the L4 kernel verification pilot project which we report on here — Kevin Elphinstone, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Ken Robinson and Adam Wiggins.

National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through *Backing Australia's Ability* and the ICT Research Centre of Excellence programs.

References

[1] J.-R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN'01, Workshop on Model Checking of Software*, volume 2057 of *LNCS*, pages 103–122, 2001.

[3] S. Berghofer and M. Strecker. Extracting a formally verified, fully executable compiler from a proof assistant. In *Proc. COCV'03*, Electronic Notes in Theoretical Computer Science, pages 33–50, 2003.

[4] W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.

[5] B. C. Brock, W. A. Hunt, Jr., and M. Kaufmann. The FM9001 microprocessor proof. Technical Report 86, Computational Logic, Inc., 1994.

[6] A. Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(2):127–139, June 1989.

[7] A. Fox. Formal specification and verification of ARM6. In D. Basin and B. Wolff, editors, *TPHOLs '03*, volume 2758 of *LNCS*, pages 25–40. Springer, 2003.

[8] S. Hallem, B. Chelf, Y. Xie, and D. R. Engler. A system and language for building system-specific, static analyses. In *PLDI*, pages 69–82, 2002.

[9] M. Hohmuth, H. Tews, and S. G. Stephens. Applying source-code verification to a microkernel — the VFiasco project. Technical Report TUD-FI02-03-März, TU Dresden, 2002.

[10] In G. Klein, editor, *Proc. NICTA FM Workshop on OS Verification*. Technical Report 0401005T-1, National ICT Australia, 2004.

[11] G. Klein and H. Tuch. Towards verified virtual memory in L4. In K. Slind, editor. *TPHOLs Emerging Trends '04*, Park City, Utah, USA, 2004.

[12] L4Ka Team. *L4 eXperimental Kernel Reference Manual Version X.2*. University of Karlsruhe, Oct 2001. `http://l4ka.org/projects/version4/l4-x2.pdf`.

[13] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. Technical Report CSL-116, SRI International, 1980.

[14] M. Norrish. *C formalised in HOL*. PhD thesis, Computer Laboratory, University of Cambridge, 1998.

[15] J. Shapiro, M. S. Doerrie, E. Northup, S. Sridhar, and M. Miller. Towards a verified, general-purpose operating system kernel. In Klein [10], pages 1–19.

[16] H. Tuch and G. Klein. Verifying the L4 virtual memory subsystem. In Klein [10], pages 73–97.

[17] P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, and G. Back. Formal methods: a practical tool for OS implementors. In *HotOS-VI*, 1997.

[18] VerifiCard project. `http://verificard.org`, 2005.

[19] VeriSoft project. `http://www.verisoft.de`, 2005.

[20] B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, 1980.

Making Events Less Slippery With *eel*

Ryan Cunningham and Eddie Kohler
University of California, Los Angeles
rcunning@gmail.com, kohler@cs.ucla.edu

Abstract

Event-driven programming divides a program's logical control flow into a series of callback functions, making its behavior difficult to follow. However, current program analysis techniques can preserve the event model while making event-driven code easier to read, write, debug and maintain. We designed the Explicit Event Library (*libeel*) to be amenable to program analysis, and created tools to graphically expose control flow, verify resource safety properties, and simplify debugging. The result sustains the advantages of event-driven programming while adding the important advantage of programmability.

1 Introduction

Coping with asynchronous events generated by unpredictable sources is a fundamental systems problem, with two fundamentally dual solutions [12]: threads and event-driven programming. Despite controversy old and new [8, 14, 18, 19], both models have their place—and in particular, event-driven programming is here to stay. In some contexts, such as interrupt handlers and embedded systems, a connection-oriented thread model doesn't fit the problem or isn't supported by underlying layers. In others, such as Web serving, event-driven programs achieve the best published performance [11, 17] and expose important information, such as blocking points [8].

Unfortunately, event-driven programs remain difficult to understand. Control flow is divided into many cooperatively-scheduled callback functions, obscuring context and programmer intent. This makes it hard to write event-driven programs and, worse, hard to analyze and debug them when they go wrong. Although threaded programs have their own difficulties, particularly with synchronization, threading doesn't obfuscate programs in the same way. So are threads the only model suitable for dependable software? Put another way, must tools for improving event-driven programmability "effectively duplicate the syntax and run-time behavior of threads" [18]?

We show that current program analysis techniques can preserve the event-driven programming model while making event-driven programs easier to read, write, debug, and maintain. We designed a simple event library—*libeel*, the Explicit Event Library—to be amenable to program analysis. All relevant arguments are presented directly to the library, rather than stored in heap structures requiring pointer analysis. Also, a *group identifier* argument encourages the programmer to group callbacks dealing with the same conceptual connection, enabling easy discovery of the program's logical control flow. With the help of this library, we built tools that graphically expose the event-driven control flow; that verify program properties, such as that all resources allocated on a path are freed; and that simplify debugging. Two programs, crawl-0.4 [15] and plb-0.3 [5], were ported to *libeel* from the *libevent* library [16]. The *eel* tools helped us understand these programs and uncovered several bugs, while preserving the advantages of event-driven programming.

Our contributions are *libeel*, an event notification library that facilitates readable programming and (through its group identifiers) easy analysis, and the *eelstatechart*, *eelverify*, and *eelgdb* tools built above it.

2 Event Programming

This section explores some typical event-driven code for fetching an HTTP document, demonstrating common problems with event-driven software's readability, writability, and debuggability. The code is in Figure 1.

First, we try to understand the code. The control path clearly proceeds from `http_fetch` to `readheadercb` following a read readiness event, or to `timeoutcb` after a timeout expiration. However, it is not clear what happens following the return on line 29. One would have to read the function `http_parseheader`, and any functions it calls, in order to determine the next callback in the chain, if any. Determining the control flow of event-driven programs often requires reading the entire function call graph to assemble the callback chain.

Determining where files, memory, and other resources are reclaimed also becomes a complicated process. Callback functions can allocate either local resources, which last only as long as the callback function itself, or long-lived resources, which are passed to the next callback as part of the connection state. Furthermore, one callback function can free resources passed to it by a prior callback. When reading the code, it's difficult to tell how resources should be categorized—and, for example, whether the absence of a "free" function represents a memory leak.

"Stack ripping" [6] makes this even worse. When a sequential, blocking function is modified to wait for an event, it must move all of its relevant state information, possibly including stack variables, to the heap structure passed to the next callback. For example, Figure 1's line 6 writes an HTTP request to a file descriptor using a nor-

```
1  // assume uri->fd is ready for write
2  void http_fetch(struct uri *uri, eel_group_id gid) {
3    char req[1024];
4    // create the HTTP request and write it to uri->fd
5    snprintf(req, sizeof(req), "%s %s HTTP/1.0\r\n" ...);
6    atomicio(write, uri->fd, req, strlen(req));
7    // wait for a read event on uri->fd or timeout
8    eel_add_read_timeout(gid, readheadercb,
         timeoutcb, uri, uri->fd, HTTP_READTIMEOUT);
9  }
10 // the timeout occurred before uri->fd was ready to read
11 void timeoutcb(eel_group_id gid, void *arg, int fd) {
12   // clean up all resources; ends the callback chain
13   uri_free_gid((struct uri *)arg, gid);
14 }
15 // uri->fd is ready to read
16 void readheadercb(eel_group_id gid, void *arg, int fd) {
17   char line[2048];
18   struct uri *uri = arg;
19   // read some data from uri->fd
20   ssize_t n = read(uri->fd, line, sizeof(line));
21   if (n == -1) {
22     if (errno == EINTR || errno == EAGAIN)
23       goto readmore; // wait for another read event
24     uri_free_gid(uri, gid); // real error: free and return
25     return;
26   } else if (n == 0) // ... handle other conditions
27   // ... copy unparsed header info into uri structure
28   http_parseheader(uri, gid);
29   return; // What callback is next???
30 readmore:
31   // wait for another read event or timeout
32   eel_add_read_timeout(gid, readheadercb,
         timeoutcb, uri, uri->fd, HTTP_READTIMEOUT);
33 }
```

Figure 1: Code from a version of crawl-0.4 [15] ported to *libeel*, showing part of a typical HTTP document fetch.

mal, non-blocking write. While this particular write is extremely unlikely to block in practice, true non-blocking I/O would require that any unused portion of `req` be passed on to the next callback.

Stack ripping complicates writing as well as reading. Consider a programmer writing Figure 1's code in top-down order. Once she finishes writing `readheadercb`, she might write `http_parseheader`. Unfortunately, this involves cleaning up some subset of `readheadercb`'s state; and whenever `readheadercb`'s state changes, `http_parseheader` must change too.

Say the programmer now wishes to debug by stepping line by line through the source code, observing variable values. She runs the program in a debugger and sets a breakpoint at line 6 to begin the process. After stepping a few lines to the end of `http_fetch`, the debugger steps to the calling function—but this is the dispatch loop. There is no convenient way to continue stepping on to the next line of logical program flow (11 or 16). Debuggers don't follow the logical control flow of event-driven programs, making stepping inconvenient.

In practice, programmers have avoided these problems primarily by turning to threads, whose explicit control flow improves programmability. Memory is more easily managed because stack variables can be used across blocking calls. Other resources are more easily managed because control paths that exit the function are more visible. Debugging is easier (assuming the debugger has thread support). Programmers that choose to use events, often for performance reasons, suffer through with ad-hoc solutions. For instance, separate documentation might be manually created to show the callback chain; memory and resource management is most likely done manually; `printf` debugging rules the day. Some systems combine events' cooperatively-scheduled execution model with thread-like code via automatic stack management [6, 19]; but this may not support multiple outstanding callbacks on the same connection, and still requires the programmer to revalidate shared state after each blocking call [6].

3 The *eel* Tools

Our *eel* tools and a library framework attack all these problems at their common source: the difficulty of following an event-driven program's control flow. The *libeel* library simultaneously facilitates event-driven programming and program analysis: we designed the library specifically to avoid the aliasing and state issues that typically complicate analysis of C-based programs. Nevertheless, *libeel* programs are truly event-driven, not event-based programs in threaded clothing.

The tools leverage *libeel* to extract control-flow information from arbitrary event-driven programs. The results are displayed or used to verify program properties. *eelstatechart* visualizes the program's control flow in the form of a simple chart. The *eelverify* framework can detect resource leaks and other mistakes common to event-driven programs. Lastly, a modified *gdb* lets the programmer transparently step through the callback chain, simplifying debugging. Each tool plays a role in the programming process: *eelstatechart* in program comprehension, *eelverify* in checking, and *eelgdb* in debugging.

The *libeel* library was initially based on *libevent* [16], another event library, although it has considerably diverged. The *eel* tools were built using the C Intermediate Language (CIL) framework for C program manipulation and analysis [2], the BLAST software verification system [1], *gdb* [4], and Graphviz's *dot* [3].

3.1 The *libeel* interface

The *libeel* library, like other existing event libraries [8, 16], provides a single unified interface for registering, canceling, and dispatching callbacks. It abstracts system dependencies, such as the choice of *select* or a more-scalable variant [7, 13]. Figure 2 shows part of its interface. The event functions register a callback for an I/O event on the given file descriptor, or for a timer that goes off after a certain number of milliseconds. Other functions combine I/O with timeout events. The design challenge was to provide a usable, minimal interface that simultaneously enables analysis.

```
// Group operations
eel_group_id eel_new_group_id(void);
void eel_delete_group_id(eel_group_id gid);
// Event functions
eel_event_id eel_add_timer(eel_group_id gid, eel_callback cb, void *cb_arg, int timeout_milliseconds);
eel_event_id eel_add_read(eel_group_id gid, eel_callback cb, void *cb_arg, int fd);
eel_event_id eel_add_write(eel_group_id gid, eel_callback cb, void *cb_arg, int fd);
eel_event_id eel_add_error(eel_group_id gid, eel_callback cb, void *cb_arg, int fd);
```

Figure 2: Some of the *libeel* interface, including showing group identifier new and delete calls and event registration functions.

libeel's interface is simpler than some other event notification libraries in that the callback functions are explicitly named for each event registration, and there is a one to one pairing of registrations and callback calls. *libeel* also requires the programmer to specify the logical connection to which an event applies, via group identifier arguments in all event registration calls. Explicit functions create and destroy group identifiers. It is typically easy to add group identifiers to an event-driven program: context data and resources passed along the call chain are usually allocated in a single location and deallocated in another; the group identifier can be created and released at these sites as well. Group identifiers somewhat resemble thread identifiers, but differ in that there can be multiple callbacks outstanding for the same group. The other *eel* tools trace group identifier values through the program's callbacks to extract logical code paths.

Initially, we considered using *libevent* directly, but doing so proved difficult. Event registration in *libevent* requires two library calls, one to set up a parameter data structure and one to actually register:

```
event_set(&ev, fd, EV_READ|EV_WRITE|EV_PERSIST,
        callback, NULL);
... // analysis must check whether ev has changed
event_add(&ev, &timeout);
```

This allows persistent (automatically recurring) registrations and multiple event types registered to the same callback function, and encourages persistent `ev` structures. For example, one `ev` might be initialized at the beginning of the program, then reused liberally throughout. Thus, whole-program alias analysis might be necessary to determine the callback function registered by a particular `event_add`, complicating both control flow analysis and human understanding.

libeel avoids these issues by requiring that all parameters be presented as explicit arguments, and by disallowing recurring registrations. The resulting one-to-one correspondence between a single event registration and a single callback firing decouples the semantic cases. These interface design differences keep the *libeel* semantics simple enough for program analysis and as flexible as *libevent* (although the latter is less verbose and marginally more efficient). Porting a *libevent* program to *libeel* is straightforward: separate out the multiple event types and persistent event registrations into independent callback functions and registrations. However, in cases where registration parameters are set distant from actual registrations (typically because the parameter structure is reused throughout the program), one must do whole program reasoning to determine what events are being registered to what callback function.

3.2 *eelstatechart*: visualizing the callback chain

eelstatechart helps *libeel* programmers better understand a program's control flow. Short of modifying C syntax in a non-trivial way, asynchronous execution can best be visualized using a graph. The chart we generate here is equivalent to the graph described by Lauer and Needham in 1978 [12] and the blocking graph described by von Behren et al. [18] Nodes in an *eelstatechart* are labeled with callback function names and edges with abbreviations for I/O or timer events. The purpose is to make the program's underlying structure more obvious, helping the programmer understand the common paths and how connections progress. Callbacks obscure even simple programs by removing context; *eelstatechart* recovers each callback's context in the program.

eelstatechart performs a static analysis; the tree of event registrations and their associated callbacks is determined while following the creation, use and release of group identifiers through the static callgraph of the program. *eelstatechart* starts by visiting all function definitions and their static function calls to build a call graph. When a *libeel* call is encountered it marks the calling function with a label indicating the operation performed. The source of the group identifier is located and added to the label as well. Finally, these labels are percolated all the way up the callgraph. To export the chart, the labels are traversed from callback to callback beginning with the program entry point.

Figure 3 shows the primary *eelstatechart* for crawl-0.4 [15], a simple Web crawler. The code from Figure 1 appears on the right side of the figure. `http_fetch` is called by `http_connectioncb`, creating the read readiness event and timeout event seen heading down and right from `http_connectioncb`. Once in `readheadercb`, the chart shows arrows indicating the callback registrations from line 31. It also shows an arrow to "delete", indicat-

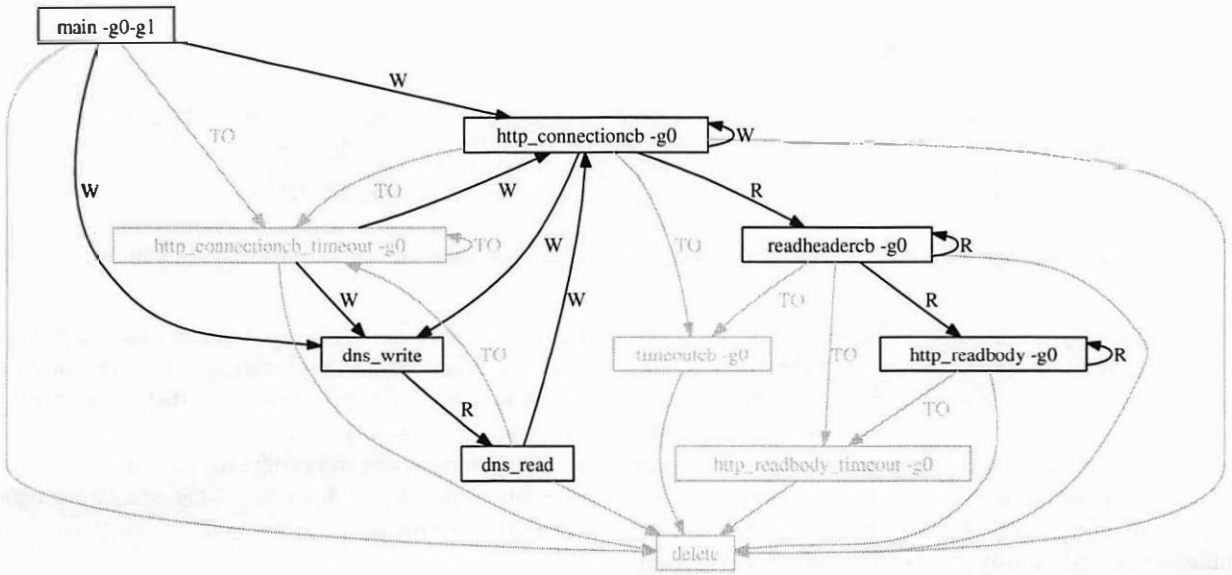


Figure 3: The primary *eelstatechart* for crawl-0.4 [15]. Each rectangle names a callback function. Each arrow indicates the next callback in the chain. Arrows are labeled with abbreviations of the event causing the callback to be fired: "W" is write, for example. Arrows pointing to "delete" indicate the end of the callback chain. Gray rectangles and arrows indicate timeout or delete paths, which typically correspond to errors.

ing that the callback chain can end, in this case from a call to `uri_free_gi` don line 24 or elsewhere. The call to `http_parseheader` on line 28 extends the callback chain to `http_readbody` or `http_readbody_timeout`, which go on to repeat back to `http_readbody` or end the chain. By just reading the code it is not apparent what callbacks might be generated inside the call to `http_parseheader`; *eelstatechart* clearly conveys this information.

eelstatechart will generate an approximation of the true chart, rather than the true chart, if an event registration uses a variable to name a callback function (rather than a naming a callback function directly), or due to complex use of function pointers elsewhere in the code. This hasn't happened in the programs we've converted so far. One remaining challenge is to create a chart that is easily read but also contains all pertinent information. For example, it would be especially nice to show what lines generated which events. We collect enough detail to provide this information, but it would clutter the chart beyond easy readability. Another challenge is visualizing cases where more than one next callback is registered, i.e. the control proceeds down both callback chains in an unspecified order—a particularly flexible pattern.

3.3 *eelverify*: a verification framework

eelverify is a framework for verifying properties of *libeel* programs. It provides a set of program transformations and instrumentation points for *libeel* programs, as well as verifiers that use these transformations. For instance, *eelverify* can verify that group identifiers are not leaked anywhere along the callback chain. It first performs a simple program transformation so that callback functions can be verified independent of each other. Then BLAST [10] is used to instrument the `eel_group_idtype`, *libeel* calls, and callback function returns such that if a group identifier is leaked, an error label is reached. Other properties can be verified using a similar approach.

Using *eelverify* we found a few actual bugs (and a few false positives) from two programs that, together, had about 15,000 lines of uncommented C code. One interesting bug stands out in plb-0.3 [5], an HTTP load balancer. The offending code segment is in a callback function, `client_forward_request`, executed following a read readiness event. It then attempts to execute the read call. On an error read result it checks for `EINTR`, which indicates that a signal interrupted the read attempt. Typically this situation is handled by waiting again for a read readiness event, but the callback simply returns without registering any callback or releasing resources. Here, `EINTR` would result in a failure to forward HTTP POST data from the client to the server. *eelverify* found this bug because the group identifier passed into the callback function was not used or released along the call path. It's worth noting that this bug might be hard for an automatic checker to detect [9]. Since different callbacks were set on different paths, some exit points deleted the group identifier, while others did not.

eelverify implicitly assumes that *libeel* is correct; it uses the *libeel* semantics but acts on its functions as if they were language keywords. *libeel* cannot be verified directly because it uses function pointers and com-

plex data structures to manage callback dispatch. Under this assumption its analysis is sound, however, meaning *eelverify* never will report a false negative. Function pointer usage inside callback functions can lead to false positives, however.

eelverify provides a framework for verifying a broader class of resource properties, including those that follow a paired calling pattern such as create/release, alloc/free, or open/close, within the context of a *libeel* event-driven program. For example, it might ensure that file descriptors are always closed after being opened, or that they are not used after being closed. However, *eelverify* can currently verify properties only along a single instance of the callback chain; it ignores any dependencies between instances or between separate chains.

3.4 Debugging with *eel*

eelgdb's extensions consist of a few new commands that allow stepping line by line through a *libeel* callback chain. 'Cnext' is similar to the *gdb* 'next' command, except that if the current line matches a pattern indicating the addition of a *libeel* event, it will create a new temporary conditional breakpoint at that callback function's header. These breakpoints will only stop the program if the group identifier argument equals that of the currently active callback. Thus, program execution can continue until the next breakpoint in the logical connection, allowing for transparent stepping to the next logical point in the program. The result is that the debugger allows callbacks for other connections to be dispatched while it is waiting for the next relevant event, but returns control to the user once an event for the current connection has triggered. The analogous situation in the threaded model is that when an I/O call blocks, the debugger executes code on other threads while it waits for the I/O call to complete.

For example, consider debugging the code:

```
1    ...
2    atomicio(write, uri->fd, req, strlen(req));
3    eel_add_read(gid1, readheadercb, uri, uri->fd, 1000);
4  }
5  void readheadercb(eel_group_id gid2, void *arg, int fd) {
6    ...
```

Assume the program is run in *eelgdb*, which hits a breakpoint on line 2. The user executes 'cnext', which causes the debugger to step to line 3, just as 'next' would. When 'cnext' is applied to line 3, a *libeel* pattern matches the line of source code, extracting the expression `gid1` and the identifier `readheadercb`. (As with the other *eel* tools, *eelgdb* does not currently handle function pointer usage in event registrations.) Next it evaluates `gid1`'s value at line 3 (e.g. `0x007A224F`) and sets a conditional breakpoint as follows: **`tbreak`** `readheadercb` **`when`** `(gid2 == 0x007A224F)`. Then it steps over line 3 to line 4. The user can then 'continue' to allow the program to proceed or step back to the calling function. Once the program is continued, if the read event is triggered on the same group identifier, the *libeel* dispatch loop will call `readheadercb` and hit the breakpoint on line 5. The user then proceeds debugging the same instance.

4 Related Work

In 1978, Lauer and Needham proved that threads and events are duals [12]. Most still researches believe that one or the other is better, however. Ousterhout argued that threads are a bad idea because they perform poorly, and concurrency issues make them error-prone [14]. Von Behren et al. argue, in contrast, that event-based programs are too difficult to write, for the reasons we have explained [18]. They aimed to improve the performance of threads to match that of events; Capriccio's compiler analyses and runtime techniques change a threaded program's runtime behavior into that of a cooperatively-scheduled event-driven program [19]. Even here, events and threads are dual: events need no compiler help for performance, since they perform well already; instead, we use analyses and *static* techniques to improve the programmability of events to match that of threads (or, arguably, better that of preemptively-scheduled threads, because there are no concurrency issues). Adya et al. named "stack ripping", identified it as a major issue with event-driven programming, and introduced a mechanism for automatically managing multiple stacks [6]. The *libeel* library leaves the user to manage the stack manually, and the existing *eel* tools address the problems that result. *Eel*-like tools for a system with automatic stack management would address its problems instead—for instance, by checking that any stack copies of global state are revalidated after each blocking call.

Several projects focus on building fast web servers, or fair web servers, using events [11, 17] or a combination of events and threads, as in SEDA [20]. Dabek et al. describe a C++ library, *libasync*, for building robust event-driven software [8]. *libasync* primarily addresses callback safety by using C++ templates to cross-check callback function types and context data. It also adds reference-counted objects to ameliorate some resource management issues. We focused on enabling and building static tools that check safety issues and facilitate program clarity; reference counting and type checking would be complementary.

5 Conclusion

The *eel* library and program analysis tools help programmers evade common problems with the event-driven model, while remaining inside that model. We are working on further improvements to visualization to differentiate success and error execution paths, and on verifying

other properties such as proper file descriptor usage. We are also working on a program transformation, in conjunction with modifications to BLAST, that would allow verification using regular BLAST specifications, instead of those phrased to verify properties of callback functions independent of each other. This would let us verify properties that require simultaneous analysis of more than one callback. As well, collecting profiling information tagged with group identifiers could aid in debugging resource bottlenecks in *libeel* programs. Even now, however, the *eel* tools make it easier to read, write, debug and maintain event-driven programs. Code will be available at `http://read.cs.ucla.edu/`.

Acknowledgements

We gratefully acknowledge Rupak Majumdar for discussions and BLAST aid, and the anonymous reviewers for helpful comments. This material is based upon work supported by the National Science Foundation under Grant No. 0427202.

References

[1] BLAST: Berkeley Lazy Abstraction Software Verification Tool. URL `http://www-cad.eecs.berkeley.edu/~rupak/blast/`.

[2] CIL—Infrastructure for C program analysis and transformation. URL `http://manju.cs.berkeley.edu/cil/`.

[3] Graphviz—graph visualization software. URL `http://graphviz.org/`.

[4] GDB: The GNU Project Debugger. URL `http://www.gnu.org/software/gdb/gdb.html`.

[5] PLB—Pure Load Balancer: A free high-performance load balancer for Unix. URL `http://plb.sunsite.dk/`.

[6] A. Adya, J. Howell, M. Theimer, B. Bolosky, and J. Douceur. Cooperative task management without manual stack management. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, June 2002.

[7] G. Banga, J. C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proc. 1999 USENIX Annual Technical Conference*, pages 253–265, Monterey, California, June 1999.

[8] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris. Event-driven programming for robust software. In *Proceedings of the 2002 SIGOPS European Workshop*, September 2002.

[9] D. Engler, D. Yu Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. 18th ACM Symposium on Operating Systems Principles*, pages 57–72, Château Lake Louise, Alberta, Canada, Oct. 2001.

[10] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, pages 235–239. Lecture Notes in Computer Science 2648, Springer-Verlag, 2003.

[11] M. Krohn. Building secure high-performance web services with OKWS. In *Proc. 2004 USENIX Annual Technical Conference*, Boston, Massachusetts, June 2004.

[12] H. C. Lauer and R. M. Needham. On the duality of operating system structures. In *Second International Symposium on Operating Systems*, pages 408–423. INRIA, October 1978.

[13] J. Lemon. Kqueue: A generic and scalable event notification facility. In *Proceedings of the FREENIX Track (USENIX-01)*, June 2001.

[14] J. K. Ousterhout. Why threads are a bad idea (for most purposes). Presentation at the 1996 USENIX Annual Technical Conference, Jan. 1996.

[15] N. Provos. crawl—a small and efficient HTTP crawler. URL `http://www.monkey.org/~provos/crawl/`.

[16] N. Provos. libevent—an event notification library. URL `http://www.monkey.org/~provos/libevent/`.

[17] Y. Ruan and V. Pai. Making the "box" transparent: System call performance as a first-class result. In *Proc. 2004 USENIX Annual Technical Conference*, Boston, Massachusetts, June 2004.

[18] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proc. HotOS-IX: The 9th Workshop on Hot Topics in Operating Systems*, Lihue, Hawaii, May 2003.

[19] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for Internet services. In *Proc. 19th ACM Symposium on Operating Systems Principles*, pages 268–281, Bolton Landing, Lake George, New York, Oct. 2003.

[20] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. 18th ACM Symposium on Operating Systems Principles*, pages 230–243, Château Lake Louise, Alberta, Canada, Oct. 2001.

Parallax: Managing Storage for a Million Machines

Andrew Warfield, Russ Ross, Keir Fraser, Christian Limpach, Steven Hand
University of Cambridge Computer Laboratory

1 Introduction and Motivation

OS virtualization is drastically changing the face of system administration for large computer installations such as commercial datacenters and scientific clusters. A recent report by Gartner predicts that commercial use of virtualization will triple over the five year period beginning in 2004 [1]. While it is commonly held that OS virtualization improves the utility, manageability, and scalability of large-scale environments, we believe that it is not sufficient in itself. In this paper we argue that the next key challenge facing these environments lies in the dramatically evolving requirements for the management of persistent storage.

More hosts: Over the past few years, academic labs, server hosting centers, banks and other related organizations have moved firmly in the direction of centralizing compute resources into single facilities. Clusters especially have gained considerable momentum: academic installations of between 500 and 1,000 nodes are not uncommon and we are aware of several industrial installations of between 5,000 and 10,000 machines in operation today. In these environments, OS virtualization will result in a multiplication by between 10 and 100 in the number of active operating system instances; we have corresponded with several organizations who expect one million virtual node clusters within the next few years. Needless to say, each one of these hosts requires a system image to boot from.

More availability: Live OS migration [2] represents a qualitative shift in the management of these systems. Virtual hosts may be moved between physical systems while they run: this not only allows administrators increased freedom to service hardware but is also being explored as a mechanism for load-balancing in cluster environments. In order for a VM to migrate, its system image must remain available, mandating the location and access transparency of persistent storage.

More history: In addition to the benefits of physical separation provided by migration, several research projects have explored the benefits available through storing historical versions of VM state and allowing them to "time-travel". In these projects, a VM is rewound to a previously checkpointed state and either resumes execution there or is replayed using an instruction trace relative to the checkpoint. Revisiting these past states of a VM's execution has been used for intrusion detection [3], configuration debugging [4], and debugging for software development [5]. For these approaches to work though, entire versions of a VM's block devices must be captured along-side the suspended memory and processor state. In extremis, it is foreseeable that enough historical state could be preserved to perform instruction-granularity replay through the entire life of a cluster. Such functionality would provide a complete set of forensic information and be of interest to highly-secure installations.

These three orthogonal issues each imply an increase in the scale of storage required for clusters of virtual machines. In this paper we propose *Parallax*, a distributed storage system which simultaneously provides different views on a single underlying distributed block store. Parallax tackles the problems of management and scale for huge numbers of both active and historical system images in large cluster environments.

The nature of this new environment has led to two key design decisions that distinguish Parallax from previous systems. First, we observe that system image management is effectively free of write sharing. This allows us to easily exploit persistent caching for high performance and to eschew the complexity of a distributed lock manager. Second, we capitalize on the nature of the virtualized environment to run an isolated Parallax server on each physical host, giving it control of local disk and allowing it to serve the set of local VMs directly. Parallax also uses block-level copy-on-write techniques to support both sharing and frequent, lightweight snapshots.

2 Design Space

An executing virtual machine requires a certain amount of persistent storage to hold a root file system, application data, swap files, and so on. Over time, VMs may wish to snapshot their persistent storage to allow backup, to deal with subsequent application or human errors, or even to allow "time-travel" as described in Section 1. Finally, there may be storage required for VMs not currently executing but which may be deployed (or re-deployed) in the future.

We unify all forms of persistent storage in a virtual server farm under the concept of a *virtual disk image* (VDI), the basic unit of management. A VDI represents the current, writable persistent state of a virtual disk, as well as a set of immutable snapshots representing the state of the VM at points in its history. A VDI is accessible from any physical machine in the cluster, and is stored in a redundant fashion to ensure high availability and durability. VDIs have human-readable site-unique names which facilitate the life-cycle management of virtual machines (e.g. deployment, snapshotting, suspension, time-travel).

It is quite reasonable to think of managing millions or even tens of millions of VDIs across a single cluster. In the following, we first discuss why existing techniques are inadequate, and then present our design for Parallax and how it addresses this challenge.

2.1 Yet another distributed storage system?

Storage systems have been one of the most exhaustively explored aspects of systems research over the past 30 years. Probably the most relevant state-of-the-art in cluster-wide image management is that of storage area networks (SANs). There are several current commercial offerings which tout "storage virtualization": systems that aggregate a set of storage servers into a single block-level substrate, and then allow this substrate to be divided up into individual volumes for export to network-attached hosts. Four important factors distinguish Parallax from these systems.

First, SANs are very expensive. Many, especially academic, environments will desire an alternative to expensive storage products. Furthermore, given that clusters are typically built from commodity systems, each housing a commodity disk, it seems reasonable to build a storage system that aggregates these disks. A virtualized environment makes this even more desirable given that the system-wide set of disks may be directly controlled using a set of per-host, isolated virtual machines. The challenge here is to provide the *manageability* afforded by SANs in this new environment.

Next, the scale that we are attempting far exceeds the capacity of any SAN that we are currently aware of. Fortunately there is an economy to this scale: we expect hosts to be based on a small set of original *template* disk images, and take advantage of the fact that common blocks may be shared across images. The underlying block store in our system will overlay common data where efficiency permits, allowing common blocks to be shared in many situations.

Third, the creation of new disk images is of critical importance to our scheme. Preserving historical images requires frequent run-time snapshotting of active OS images. A design goal that we are targeting is to be able to efficiently snapshot a running OS's disk and memory state every thirty seconds. Additionally, we anticipate that new virtual machine instances will generally be composed from existing templates, and so the duplication of VDIs is also important. A fundamental aspect of our design is in the management of per-VM block metadata, and providing fast primitives to fork and snapshot an active image.

Finally, we make the observation that write sharing is unnecessary in VDI management since at any given time, there is at most a single VM associated with a particular VDI. We take advantage of this fact to aggressively write-optimize our system, and achieve very high disk performance with considerably less complexity than is seen in systems using a distributed lock manager and lease-based persistent caching.

2.2 Parallax: Basic design

Our basic approach is to eliminate write-sharing, enable aggressive client-side persistent caching, seed the system with a small number of template images, use snapshot and copy-on-write to allow block-level sharing and use simple replication for high availability and durability.

The local storage on each physical machine is partitioned into a persistent cache for locally hosted VMs and a contribution to a pool of distributed storage shared by the cluster. These two tasks are provided as a service running in an isolated "Parallax VM" that presents a simple block device abstraction to each user VM and translates requests for the virtual blocks that are visible to the VMs into requests for physical blocks distributed throughout the cluster.

Each virtual disk is described in metadata as a log of snapshots, each pointing to the root of a radix tree. Radix trees allow an effecient copy-on-write representation of mappings from virtual disk blocks to 64 -or 128-

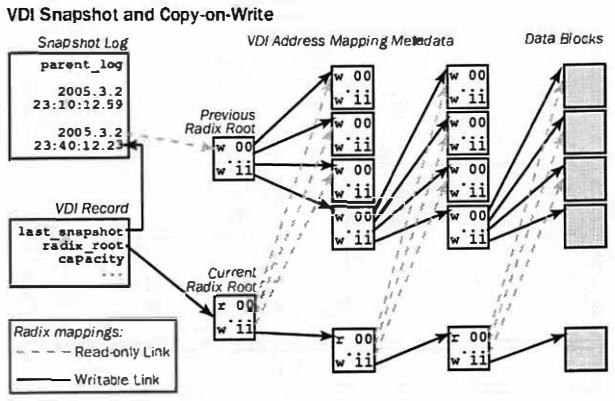


Figure 1: VDI Snapshot and Copy-on-Write

bit cluster-unique physical block identifiers. All but the last radix tree in a snapshot log are immutable, and the mutable tree is only written to by a single VM, allowing common blocks to be shared across images without requiring distributed mutual exclusion.

This approach makes the creation of both snapshots and entire new VDIs both very simple and efficient operations: Figure 1 illustrates how the radix tree block mapping structure provides snapshots and copy-on-write block access for VDIs. The figure shows a simplified radix tree mapping six-bit block addresses with two address bits per radix page. The example shows a VDI that has had a snapshot taken, and successively written to a block of data at virtual block address `0x111111`.

Creating an entirely new VDI from a template image is similar to taking a snapshot. The key difference is that a new snapshot log is created, refering bak to the template snapshot as a parent. This results effectively in a fork of the parent snapshot log, allowing a new writeable radix root. We envision that a the system would initially be seeded with a set of well-known base images (Fedora Core, FreeBSD, etc.), and that new VDIs would be created based on these to serve individual VMs.

Read-only sharing is achieved for all data derived from a common ancestor image, but coincidental redundancy—e.g. where two VMs install the same package on their respective VDIs and hence create a set of duplicate blocks—is not exploited nor detected in this scheme.

Writes are generally committed first to the local disk in the persistent cache and then to the permanent replicas within the cluster. Both data blocks (parts of VDIs) and index blocks (parts of the radix tree) are persistently cached, with a subset of both also being cached in memory. The cache maintains both the virtual and physical block address for data blocks, hence avoiding the need to do the radix tree lookup for cache hits.

The persistent cache additionally serves to reduce the load on distributed storage servers. As mentioned above, a major concern in the deployment of VMs in large clusters is the greatly increased load on storage servers. The local cache serves to aggregate common read requests across the set of local VMs, lessening the load on storage servers. Write-back is performed periodically, and is also explicitly triggered by the creation of a snapshot. The frequency with which writes are pushed out from local cache to distributed storage allows administrators to trade-off data resilience and availability against load on storage servers.

Physical blocks are striped across a replication group composed of storage volumes on other hosts. Each storage server explicitly manages block allocation for its volumes. A block write to a replication group receives the allocated block ids from each server in the group and combines these ids to build the global block id for the replicated block.

2.3 Parallax: Improved sharing

Block-level snapshots with copy-on-write semantics allow extensive sharing between VMs with a common ancestor, and between historical snapshots within each individual VDI. Additional sharing of redundant content is possible if blocks are indexed by content.

The basic design can be extended to collapse redundant blocks without changing the fundamental structure of the block store and without affecting read performance and semantics. As described, the basic system uses a radix tree to map the per-VDI block numbers to universal block IDs. With the introduction of a distributed service mapping content hashes to universal block IDs, an extra step in the block write process can consolidate duplicate blocks.

Writes are made initially to the persistent cache and a content hash is computed asynchronously. This keeps potentially slow operations like hashing and collision detection out of the critical performance path. The hash is computed and the hash-to-block map is consulted to determine if the block is a duplicate. If it is, then the existing block ID is stored in the radix tree; otherwise the block is written as in the basic design and the hash-to-block map is updated.

The level of indirection for combining duplicate content allows it to be a straightforward add-on to the base architecture with the same distributed block storage pool. The look-aside cache hides most of the performance impact for writes, and nothing changes for reads. Potential storage savings are obtained at the cost of computing

content hashes and the storage and network overhead of maintaining the hash-to-block map.

2.4 The Parallax VM

An additional unique aspect of the Parallax design is that the service is hosted in an isolated VM, with direct control of local disks. Unlike historical approaches to distributed storage, and distributed services in general, this model allows centralized administration of the cluster-wide service down to the device level, and also introduces *fate sharing* between the client and server.

As a cluster-wide storage service, Parallax is a distributed conglomeration of a set of per-host storage servers, each running in an isolated VM. These VMs are given direct control of the physical disks used by Parallax: they run the physical device drivers, and export a generic block interface to local VMs accessing VDIs. This approach allows the administration of storage service within the cluster to be isolated from other administrative tasks. Administrators are free to log in to storage VMs, potentially upgrading software (even OS and device driver binaries) without requiring specific access to client VMs or to VMM management functions.

We have previously demonstrated that hosting device drivers in isolated VMs improves robustness and the ability to very quickly restart crashed driver VMs [6]. The approach described here takes this model one step further, incorporating the storage service and using the VM container to provide both performance and administrative isolation. Moreover, hosting the Parallax server on the same physical host as the clients provides a degree of fate sharing between the two. The server has the benefit of not needing to consider failures such as network partitions between it and its clients, allowing simpler fault tolerance. While the distributed storage system must still address such issues across nodes, this fate sharing provides a clean architectural interface between client VMs and the Parallax server.

We feel that this aspect of the Parallax design is a good demonstration of how VMM-based systems may be structured to avoid *liability inversion* [7]. Parallax is providing a critical system service for a set of VMs, but is not a function of the VMM itself. If the Parallax server crashes completely, only the client VMs will be affected: the remainder of the system including the VMM and the non-dependent VM instances will be completely unaffected. Further resiliency could potentially be achieved by dividing the Parallax server into separate instances, in situations where a very high degree of isolation between VMs is desired.

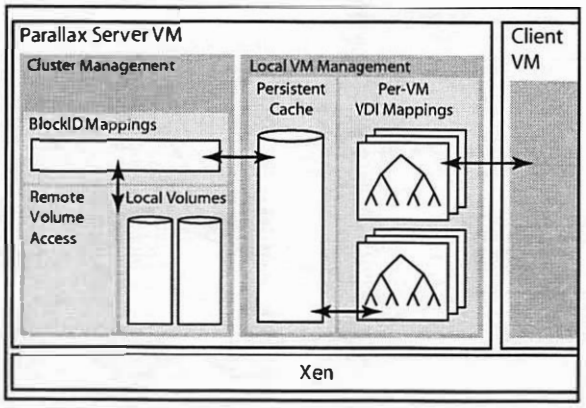


Figure 2: The Parallax server VM

2.5 Discussion

Parallax comprises a flexible and lightweight snapshot mechanism and a simple (and largely orthogonal) distributed block store for replication and enhanced availability. Provided that a sufficiently rich set of base images is provided, most of the sharing between different VMs and different generations of a single VM will be captured through common ancestry.

Duplicate content within a single image and duplicate blocks created independently in different images can be exploited by the use of content hashing. However this adds an additional mapping structure and associated computation and storage overhead: it remains to be seen whether the benefits outweigh the costs.

3 Prototype Implementation

To elucidate the design of our system, we have developed a prototype implementation over the past several months. This is not a finished artefact, but serves as a proof of concept which uses the same data paths from VM to physical disk, and allows experimentation with the various design options and techniques that we have developed.

Our prototype extends the *block tap* [8], which is a block interpositioning mechanism for the Xen VMM [9]. The block tap handles disk requests for a collection of virtual machines by forwarding them to a user-space library in an isolated VM. The tap maintains good performance while allowing us to easily modify the Parallax code.

The Parallax server is implemented as a user-space application in an isolated VM. In this configuration it is able to aggregate block requests from VMs on the local physical host and concurrently serve requests from remote hosts. The VM receives direct physical access to local storage,

and uses a GNBD[1] client library to access remote blocks.

The structure of our implementation is shown in Figure 2. The server currently implements a simple copy-on-write scheme, allowing remote GNBD images to be accessed by local VMs with writes stored on the local disk. While this implementation is considerably simpler than the full Parallax design, it serves to validate our approach and allow us to obtain baseline performance figures.

As shown in the figure, our prototype contains two points at which blockIDs are remapped. First, virtual IDs visible to VMs are mapped to a *logical ID* used by the cluster-wide block store. Second, these logical IDs are mapped to the physical hosts, disks, and blocks where the data is stored. In our prototype, this second mapping is one-to-one: VMs see the actual block addresses of a remote GNBD-mounted image. The first mapping, however, reflects the replacement of remote blocks in the VM's image with locally-stored copy-on-write blocks.

The intention of our prototype has been to guide design decisions and establish the feasibility of our approach for constructing a real system. To this end, we have measured the current performance, achieving remote read throughputs of 15MB/s to GNBD-connected images and 50MB/s to the local disk. Our implementation currently does not benefit from persistent caching, replication or parallel I/O, and uses a heavyweight mechanism to store the virtual to logical block mappings in lieu of radix trees. We are working on integrating these mechanisms into our prototype and anticipate dramatic performance improvements.

A further avenue of investigation involves the evaluation of the performance and functionality of our snapshotting and time-travel capabilities. As our design caters specifically to the frequent snapshotting of VDIs, we expect to achieve very good performance.

4 Related & Future Work

Distributed file systems have existed for over 30 years, and have been in common use since the late 80's. Most successful systems (e.g., AFS [10], NFS [11]) have in practice been 'networked file systems' in which one or a few servers export disjoint and non-replicated file systems to a number of clients. Many researchers have also proposed fully distributed file systems (e.g. Echo [12], xFS [13] and Farsite [14] to name but a few).

Our design is motivated by previous work on distributed block-level storage, most notably Petal [15] and the Federated Array of Bricks (FAB) [16]. FAB has recently also explored approaches to image snapshots [17]. Our assumption of single-writer access allows us to eschew much of the complexity present in these projects: we hope that this will allow us considerably more room to scale both in terms of number of images and frequency of snapshots.

Although we are not aware of any work directly addressing the same problem as Parallax, there are certainly similarities with other research. Frisbee [18] has explored the transport issues associated with efficiently deploying a template image onto the disks of a large number of clustered hosts. The notion of using an immutable store with copy-on-write stems back at least to Plan 9 [19], and similar techniques have been used by Elephant [20] and Venti [21]. Our current design is most similar to those from Bell Labs in that we have not considered deletion. However we hope to investigate ways in which deletion can safely be done, both to save space and to aid incremental addition and removal of storage devices.

In the future we hope to investigate how to most efficiently manage live migration [2] in the presence of aggressive persistent caching. A simple design would simply require write-back of all cached bocks for a particular VDI before a migrated VM can begin execution, but this could adversely impact VM downtime.

Instead we plan to keep LRU statistics for cached blocks on a per VM basis, allowing us to proactively transfer "hot" blocks to the destination node during live migration. Liaising with the guest operating system may also be of value, since certain blocks will already be contained within its private buffer cache. A further interesting question is whether we can choose the destination for migration based on the similarity of blocks cached at both locations; probabilistic similarity metrics such as bloom filters or sketches may make sense in this context.

Finally, we also intend to produce complete implementations of both the basic design of Parallax and the content-mapped variant, and perform extensive comparisons in terms of performance, availability guarantees, and sharing characteristics.

5 Conclusion

Virtual server farms and their variants are emerging as the architecture of choice for utility computing, and present a rather different set of distributed storage challenges. We believe Parallax represents a first step at addressing these requirements, and hope to see it evolve into the solution for these environments.

[1] `http://sources.redhat.com/cluster/gnbd`

References

[1] T. Bittman. Predicts 2004: Server virtualization evolves rapidly. *Gartner*, November 2003.

[2] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. USENIX Symposium on Networked Systems Design and Implementation*, 2005.

[3] S. T. King and P. M. Chen. Backtracking intrusions. In *Proc. 19th ACM Symposium on Operating Systems Principles*, pages 223–236, 2003.

[4] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *Proc. 6th Symposium on Operating System Design and Implementation*, pages 77–90, 2004.

[5] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proc. USENIX Annual Technical Conference*, pages 1–15, 2005.

[6] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williams. Safe hardware access with the xen virtual machine monitor. In *Proc. 1st Workshop on Operating Systems and Architectural Support for on demand IT Infrastructure (OASIS)*, October 2004.

[7] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, and D. Magenheimer. Are virtual machine monitors microkernels done right? In *Proc. 10th Workshop on Hot Topics in Operating Systems (HotOS X*.

[8] A. Warfield, K. Fraser, S. Hand, and T. Deegan. Facilitating the development of soft devices. In *Proc. USENIX Annual Technical Conference*, pages 379–382, 2005.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. 19th ACM symposium on Operating Systems Principles*, pages 164–177, 2003.

[10] J. H. Howard. An Overview of the Andrew File System. In *Proc. USENIX Winter Technical Conference*, pages 23–26, February 1988.

[11] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: Design and implementation. In *Proc. USENIX Summer Conference*, pages 137–152, 1994.

[12] A. D. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart. The Echo distributed file system. Technical Report 111, Digital Systems Research Center, October 1993.

[13] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *Proc. 15th ACM Symposium on Operating System Principles*, pages 109–126, December 1995.

[14] A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. 4th Symposium on Operating Systems Design and Implementation*, pages 1–14, December 2002.

[15] E. Lee and C. Thekkath. Petal: Distributed virtual disks. In *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, 1996.

[16] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. Fab: building distributed enterprise disk arrays from commodity components. *SIGOPS Oper. Syst. Rev.*, 38(5):48–58, 2004.

[17] M. Ji. Instant snapshots in a federated array of bricks. Technical Report HPL-2005-15, HP Laboratories, 2005.

[18] M. Hibler, L. Stoller, J. Lepreau, R. Ricci, and C. Barb. Fast, scalable disk imaging with frisbee. In *Proc. USENIX Annual Technical Conference*, 2003.

[19] S. Quinlan. A Cached WORM File System. *Software Practice and Experience*, 21(12):1289–1299, 1991.

[20] D. Santry, M. Feely, N. Hutchinson, A. Veitch, R. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 110–123, 1999.

[21] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proc. 1st USENIX Conference on File and Storage Technologies*, pages 89–101, 2002.

Stupid File Systems Are Better

Lex Stein
Harvard University

Abstract

File systems were originally designed for hosts with only one disk. Over the past 20 years, a number of increasingly complicated changes have optimized the performance of file systems on a single disk. Over the same time, storage systems have advanced on their own, separated from file systems by the narrow block interface. Storage systems have increasingly employed parallelism and virtualization. Parallelism seeks to increase throughput and strengthen fault-tolerance. Virtualization employs additional levels of data addressing indirection to improve system flexibility and lower administration costs. Do the optimizations of file systems make sense for current storage systems? In this paper, I show that the performance of a current advanced local file system is sensitive to the virtualization parameters of its storage system. Sometimes random block layout outperforms smart file system layout. In addition, random block layout stabilizes performance across several virtualization parameters. This approach has the advantage of immunizing file systems to changes in their underlying storage systems.

1 File Systems

The first popular file systems used local hard disks for persistent storage. Today there are often several hops of networking between a host and its persistent storage. Most often, that final destination is still a hard disk. Disk geometry has played a central role in the past 20 years of file system development. The first file system to make allocation decisions based on disk geometry was the BSD Fast File System (FFS) [5]. FFS improved file system throughput over the earlier UNIX file system by clustering sequentially accessed data, colocating file inodes with their data, and increasing the block size, while providing a smaller block size, called a fragment, for small files. FFS introduced the concept of the cylinder group, a three-dimensional structure consisting of consecutive disk cylinders, and the basis for managing locality to improve performance. After FFS, several other advances further optimized file system layout and access for single disks.

Log-structured file systems [7] [8] take a fundamentally different approach to data modification that is more like databases than traditional file systems. An LFS updates copy-on-write rather than update-in-place. While an LFS looks very different, its design is motivated by the same assumption as the FFS optimizations. That is, sequential operations have the best performance. Advocates of LFS argued that reads would become insignificant with large buffer caches. Using copy-on-write necessitates a cleaner thread to read and compact log segments. The behavior of log-structured file systems is still incompletely understood and the subject of ongoing research.

Journaling [3] is less radical than log-structuring and is predicated on the same assumption that sequential disk operations are the most efficient. In a log-structured file system, a single log stores all data and metadata. Journaling stores only metadata intent records in the log and seeks to improve performance by transforming metadata update commits into sequential intent writes, allowing the actual in-place update to be delayed. The on-disk data structures are not changed and there is no cleaner thread. Soft updates [2] is a different approach that aims to solve the same problem. Soft updates adds complexity to the buffer cache code so that it can carefully delay and order metatadata operations.

These advances have been predicated on the efficiency of sequential operations in a block address space. Does this hold for current storage systems?

2 Storage Systems

File systems use a simple, narrow, and stable abstract interface to storage. While the underlying system imple-

menting this interface has changed from IDE to SCSI to Fibre Channel and others, file systems have continued to use the put and get block interface abstraction. File and storage system innovation have progressed independently on the two sides of this narrow interface. While file systems have developed more and more optimizations for the single disk model of storage, storage systems have evolved on their own, and have evolved substantially from that single disk.

The first big change was disk arrays and, in particular, arrays known as Redundant Arrays of Inexpensive Disks (RAID). A paper by Patterson and Gibson [6] popularized RAID and outlined the beginnings of an imperfect but useful taxonomy called RAID levels. RAID level 0 is simply the parallel use of disks with no redundancy. Arrays employ disks in parallel to increase system throughput. They typically stripe the block address space across their component disks. For large stripes, blocks that are together in the numerical block address space will most likely be located together on the same disk. However, a file system that locates blocks that are accessed together on the same disk will prohibit the storage system from physically operating on those blocks in parallel. For a file system that translates temporal locality to numerical block address space proximity two opposing forces are in struggle. First, an increasing stripe unit will cluster blocks together and improve single disk performance. Second, an increasing stripe unit will move blocks that are accessed together onto the same storage device, reducing the opportunity for mechanical parallelism.

Storage virtualization is just a level of indirection. A translation layer does not come for free. Why are storage systems becoming increasingly virtualized? What problem is this solving? Virtualization abstracts the block address space to hide failures and facilitate transparent reconfiguration. By hiding failures, the system can use more components to achieve higher throughput, as with arrays. By allowing for transparent reconfiguration, the system can both reduce administration costs and increase reliability by allowing administrators to upgrade systems without notifying applications. Administrators can install new storage subsystems, expand capacity, or reallocate partitions without affecting file system service. The indirection of virtualization is great for storage system scalability and administration, but it completely disrupts the assumption that there is a strong link between proximity in the block address space and lower sequential access times through efficient mechanical motion.

From the outside, a storage system's virtualization looks like one monolithic map. On closer inspection, it is a layering of mappings that compose to take an address from the file system down to where it actually represents a location in physical reality. At each translation level, logical addresses are exported up and physical addresses are sent down.

```
disk:
 paddr = tbl[baddr / chunk_sz]
           + baddr % chunk_sz
array:
 (diskno, paddr) =
   fun(stripe_unit, numdisks, baddr)
volume:
 (diskno, paddr) =
   fun(stripe_unit, numdisks,
     tbl[baddr / chunk_sz]
       + baddr % chunk_sz)
```

Figure 1: **The virtualization models**
This figure shows pseudocode for the disk, array, and volume virtualization models. These models map the block addresses used by the file system to the physical addresses used internally by the storage system.

Virtualization is present in SCSI drives, where firmware remaps faulty blocks. However, at least in young and middle-aged drives, this remapping is not believed to be significant enough to meaningfully disrupt the assumptions of local file systems. One set of experiments in this paper investigates how a particular model of disk remapping affects performance. On a scale larger than single disks, virtualization is used to provide at least one level of address translation between the file and storage systems. Arrays remap addresses for striping and volumes further remap partitions across devices.

Figure 1 shows the model of virtualization used in this paper. The `baddr` is the block address used by the file system and the `paddr` and `diskno` are, respectively, the physical sector address and disk number used within the storage system. The virtualization layer maps file system block addresses to storage system physical sector addresses. The `chunk_sz` is the size of the virtualization chunk, in sectors. Chunks are remapped between virtual and physical address spaces maintaining the ordering of their internal sectors. Likewise, the `stripe_unit` is the size of the stripe and stripes are remapped maintaining their internal sector ordering. Chunking of volumes requires memory to store the individual mappings. Here this is represented as a table, `tbl`. The table is indexed into using integer division on block addresses to number chunks from base 0. The physical addresses in the table represent the base of the chunk. The block address modulo the chunk size is added to this base for the physical sector address. The deterministic remapping of striping can be computed with a function, shown here as `fun`. This function takes the stripe unit, the size of the array, `numdisks`, and the block address and outputs the sector and disk index.

3 Experimental Methodology

This paper is motivated by the question: how do file systems optimizations affect their performance on virtualized storage systems?

To answer this question, I traced two applications (macrobenchmarks) on Ext3 to generate two sets of block traces; the actual and the actual with locality optimizations destroyed but meaning preserved. Ext3 is a contemporary file system that incorporates advances such as journaling, clustering, and metadata grouping. I ran both sets of traces on a storage system simulator, varying system scale and the virtualization parameters. Sequential and random microbenchmark experiments give insight into how a randomized access pattern can outperform sequential and also stabilize performance.

All the trace generation experiments were run on the same Linux 2.6.11 system. Throughout the tracing, the configuration of the system was unchanged, consisting of 32K L1 cache, 256K L2 cache, a single 1GHz Intel Pentium III microprocessor, 768MB DRAM, and 3 8GB SCSI disks. The disks are all Seagate ST318405LW and share the same bus. One of the disks was dedicated to benchmarking, storing no other data and used by no other processes. A separate disk was used for trace logging.

I wrote in-kernel code to trace the sector number, size, and I/O type of block operations. Trace records are stored in a circular kernel buffer. A user-level utility polls the buffer, extracting records and appending them to a file on an untraced device. I traced two benchmarks; postmark and build.

Postmark [4] is a synthetic benchmark designed to use the file system as an email server does, generating many metadata operations on small files. Postmark was run with file sizes distributed uniformly betwen 512B and 16K, reads and writes of 512B, 2000 transactions, and 20000 subdirectories.

Build is a build of the kernel and modules of a preconfigured Linux 2.6.11. It is a real workload, not a synthetic benchmark. Both postmark and build start with a mount so that the buffer cache is cold.

The original postmark and build traces were generated from running their benchmarks on an Ext3 file system. I refer to these two original traces as the *smartypants* traces because Ext3 is quite clever about laying out blocks on disk.

I generated *stupid* traces by applying a random permutation of the block address space to the smartypants traces. This maintains the meaning of blocks while destroying the careful spatial locality of a smartypants file system.

Figure 2 shows the breakdown of the block traces. One result of this paper is that the stupid traces can have

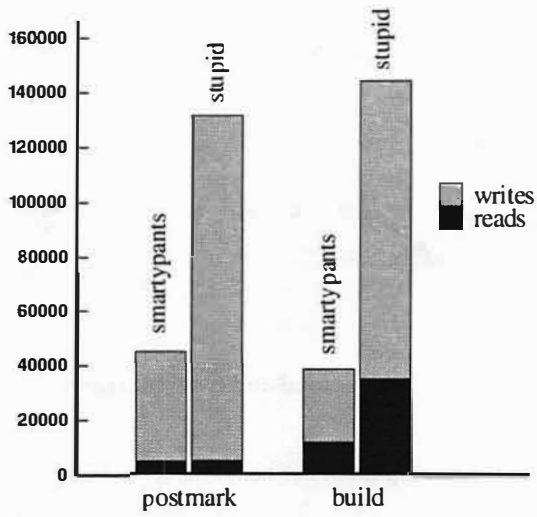


Figure 2: **Breakdown of Trace I/Os**
This figure shows the total number of I/Os for the 4 traces used in this paper. The figure breaks these totals down into their read and write categories. All stupid I/Os are done in file system blocks of 8 sector units.

competitive or even better performance. This is surprising when we look at this figure and contemplate just how deeply I brutalized smartypants to generate stupid. Not only are the I/Os of stupid scattered all over the place with absolutely no regard for interblock locality, but there are many more of them. There are many more of them because stupid only does I/O in units equal to the file system block size of 8 sectors. The workloads are dominated by writes at the block level. The read to write ratio here does not represent the ratio issued by the application because the buffer cache absorbs reads and writes.

Throughout this paper, a sector is 512B and a file system block is 8 sectors (4KB). The ratio of a particular I/O type's stupid to smartypants bar height represents the average I/O size of that smartypants I/O type measured in file system blocks. This is because stupid issues I/Os only in the size of file system blocks. For example, smartypants postmark reads are on average approximately equal to the size of a file system block, while smartypants build reads average over two file system blocks.

All experimental approaches to evaluating computer systems have their strengths and weaknesses. Trace-driven simulation is one kind of trace-driven evaluation. The central weakness of trace-driven evaluation is that the workload dows not vary depending on the behavior

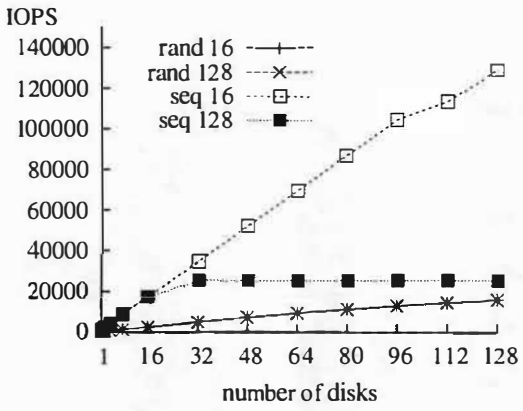


Figure 3: **sequential and random reads**

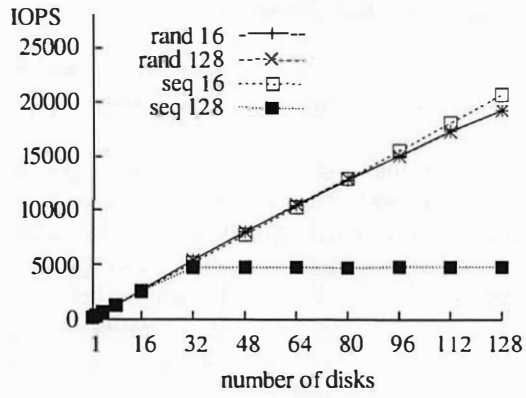


Figure 4: **sequential and random writes**

of the system. On the other hand, its central strength is that it represents something meaningful. A recent report by Zhu et al. discusses these issues [9].

I built a simulator to evaluate the performace of these workloads on a variety of storage systems. The simulator simulates arrays and uses a disk simulator, CMU DiskSim [1], as a slave to simulate disks. CMU DiskSim is distributed with several experimentally validated disk models. The experimental results reported in this paper were generated using the validated 16GB Cheetah 9LP SCSI disk model.

I used a storage simulator for two reasons. First, it allowed me to experiment with systems larger than those in our laboratory. Second, it eased the exploration of the virtualization parameter space.

The simulator implements a simple operating system for queueing I/Os. It sends the trace requests to the storage system as fast as it can, but with a window of 200 I/Os. A window size of 1 would allow no parallelism while an infinite window would neglect all interblock dependencies. Using a window size between 1 and infinity allows some I/O asynchrony without tracing interblock dependencies and without wildly overstating the opportunities for parallelism.

4 Experimental Results

The microbenchmarks are simple access patterns running directly on top of the simulator. The macrobenchmarks are all generated using the trace-driven simulation approach described in the previous section.

4.1 Microbenchmarks

The 4 microbenchmarks are read or write access with sequential or random patterns. These were run on RAID-0 arrays of varying size. The sequential read and write benchmarks issue I/Os sequentially across the block address space. The random read and write benchmarks issue I/Os randomly across the block address space. In every benchmark, I/Os are done to the storage system in 8 sector units. The microbenchmarks were run for different stripe units. Figures 3 and 4 show the results. The numbers in the figure keys are the stripe units.

Consider the read results of figure 3. Sequential far outperforms random for the smaller stripe unit of 16 sectors. This is due to track caching on the disk. Sequential rotates across the disks. When it recycles, the blocks are already waiting in the cache. For the smaller stripe units, the track cache will be filled for more cycles. As the stripe unit increases, the benefit of the track caching becomes less and less of a component, bringing the sequential throughput down to the random throughput.

Now consider the write results of figure 4. Writes do not benefit from track caching. Without the track caching, sequential and random writes have similar performance for small stripe units across array sizes. As the stripe unit increases, sequential I/Os concentrate on a smaller and smaller set of disks. Random performance is resilient to the stripe unit in both microbenchmarks. These results show how performance can be stabilized across different levels of virtualization by removing spatial locality from the I/O stream. Additionally, these results show that sometimes random access can outperform sequential by balancing load and facilitating parallelism.

4.2 Macrobenchmarks

In this section, I will discuss the results of running the traces on 3 systems; a single disk varying chunk size, an array varying the number of disks and stripe unit, and a volume varying the number of disks, the stripe unit, and the chunk size.

Figure 5 shows the four traces on a single disk. The y-axis is normalized time. The build results are normal-

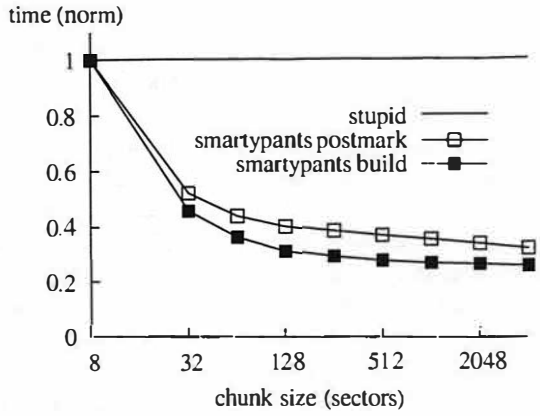


Figure 5: **Build and postmark on a single disk**
The time of postmark and build are normalized separately to the time of their stupid run for a chunk size of 8 sectors. Stupid postmark and build are indistinguishable (varying at most 1.7% for build and 0.4% for postmark) and are shown with one line.

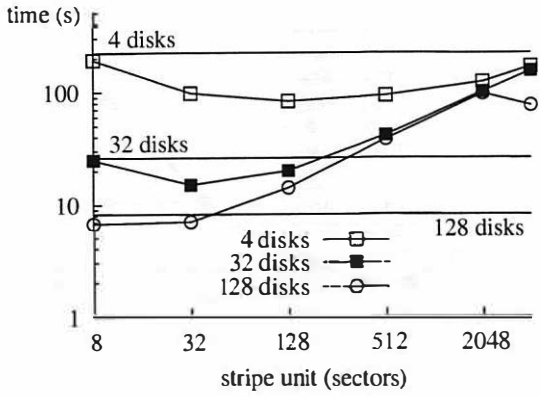


Figure 6: **Postmark on disk arrays**
The stupid results are indistinguishable for a given array size (varying at most 0.9%) and are shown with one line for each array size. The 3 smartypants results are shown as lines with points. The key shows their array sizes.

ized to the time of stupid with a chunk size of 8 sectors. Similarly, the postmark results are normalized to their stupid time with a chunk of 8. Both of the stupid lines do not vary enough to appear independent on this chart, so only one line is shown. As the chunk size gets smaller, the granularity of the virtualization remapping becomes finer, destroying the correspondence between locality in the block address space and physical locality on the disk. When the chunk is equal to the file system block size, the stupid and smartypants traces perform the same. As the chunk size increases, locality in the virtual address space begins to correspond to locality on disk across larger and larger extents. The assumptions of the smartypants optimizations begin to be true and the performance of the smartypants traces both improve by over 60% by a chunk size of 2048 sectors. This shows that all that work on improving local file system performance was not for nothing.

The stability of stupid is not limited to the single disk. You will see this in the array and volume results. Here, however, stupid is worse than smartypants. When a file system is composed into a hierarchical system its stability contributes to the total system stability. A system that values stability over performance might even prefer the stupid approach for a single disk.

Figure 6 shows the performance of the postmark traces on a RAID-0 array varying the number of disks and the stripe unit. The performance of stupid is stable across the stripe units for all 3 array sizes. Stupid scales better than smartypants. For the smaller array of 4 disks, smartypants beats stupid across all of the experimental stripe unit values. By 32 disks, stupid is beating smartypants for stripe units greater than 128 sectors. By 128 disks, smartypants performs better only for the stripe units of 8 and 32 sectors.

The down and up curve of 4 disk smartypants is seen repeatedly in the array and volume experiments. As the stripe unit increases, smartypants benefits from more efficient sequential I/O. As the stripe unit increases even further, the locality clustering of smartypants creates convoying from disk to disk as smartypants jumps from one cluster to another, bottlenecked on the internal performance of some overloaded disks, while some others remain idle. I ran the same set of experiments with the build traces and the result pattern similar.

Figure 7 shows the performance of build on a 128 disk volume that remaps chunks of a RAID-0 array using the volume model of figure 1. The experiments vary the chunk size and stripe unit across 6 stripe units and 9 chunk sizes. The 54 stupid points form an approximate flat plane with stable performance. This plane is shown here as a line. Stupid outperforms smartypants for all those configurations with stripe unit and chunk size strictly greater than 128 sectors. In this set of experiments, smartypants curves down and up across both chunk size and stripe unit. The curve breaks off into stability whenever the independent parameter exceeds the fixed one. That is because the smaller virtualization parameter dominates the remapping and neutralizes the larger one. I ran the same set of experiments with the postmark traces and the result pattern was similar.

Continuing to look at figure 7, consider the smart build trace with stripe of 32 sectors and chunk of 512 sectors. The system processes this trace at an average rate of 115.98 IOPS per disk with that average computed using a sampling period of 10ms. This average has a

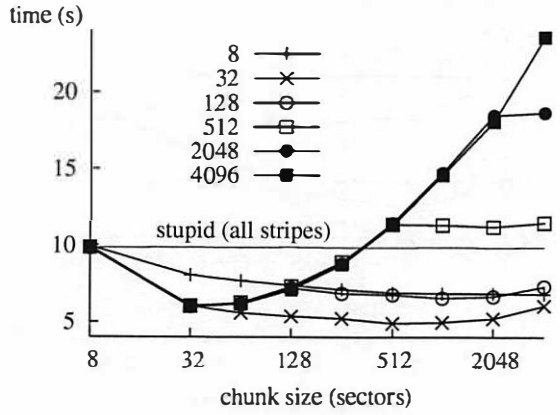


Figure 7: **Build on a 128 disk volume**
Each line varies the chunk size for a fixed stripe unit. Again, the stupid results are indistinguishable across both stripe unit and chunk size (varying 2% across all 54 points). This plane is shown as a line. The 6 smartypants lines are shown with points and the key shows the stripe unit for each.

standard deviation of 9.30% across the 128 disks. The stupid trace on the same system runs at an average rate of 135.11 IOPS per disk with a standard deviation of 7.54% across disks. As you know from figure 2, the stupid build trace consists of over 100,000 more I/Os than the smartypants build trace. The full trace takes longer even though the stupid system is able to process marginally more IOPS per disk and balance load more smoothly. For both benchmarks, the average size of the stupid I/Os is smaller than that of smartypants. Continuing to look at figure 7, now consider the smart trace with stripe and chunk of 4096 sectors. The system processes this trace at an average crawl of 7.05 IOPS per disk with a mammoth standard deviation of 58.42% across disks. During many sampling periods some disks were completely idle while others were overloaded. The stupid trace on the same system runs at an average rate of 133.82 IOPS per disk with a standard deviation of 7.60%. In this case, even though the stupid trace is much longer, it outperforms the smartypants trace. The large standard deviation of smartypants shows how smartypants layout can outsmart itself and defeat parallelism by creating overloaded hotspots.

5 Conclusions

The random permutation of the stupid traces scramble and destroy the careful block proximity decisions of smartypants. From the perspective of smartypants, virtualization also acts as a destructive permutation, though less thoroughly and with greater structure than stupid. Storage virtualization facilitates scalability, fault-tolerance, and reconfiguration, and is therefore unlikely to go away. This paper gives you two take-away results. First, I have shown that under some workloads and virtualization parameters, random layout can outperform the careful layout of a file system such as Ext3. In some cases, random layout can help a system benefit from disk parallelism by smoothly balancing load. The second, and I believe more interesting, result is how differently stupid and smartypants respond to varying virtualization parameters. In these experiments, stupid is always stoically stable while smartypants fluctuates hysterically. Data and file system images can long outlive their storage system homes. I propose random layout as a technique to immunize file systems from the instabilities of storage system configuration.

6 Acknowledgments

Discussion with Margo Seltzer inspired this paper. Discussion with Daniel Ellard refined and focused it.

References

[1] BUCY, J. S., GANGER, G. R., AND CONTRIBUTORS. The disksim simulation environment version 3.0 reference manual. Tech. Rep. CMU-CS-03-102, CMU, January 2003.

[2] GANGER, G. R., AND PATT, Y. N. Metadata update performance in file systems. In *Proceedings of the USENIX 1994 Symposium on Operating Systems Design and Implementation* (Monterey, CA, USA, November 1994), pp. 49–60.

[3] HAGMANN, R. B. Reimplementing the cedar file system using logging and group commit. In *Proceedings of the eleventh ACM Symposium on Operating Systems Principles* (Austin, TX, USA, November 1987), pp. 155–162.

[4] KATCHER, J. Postmark: A new filesystem benchmark. Tech. Rep. TR3022, Network Appliance, 1997.

[5] MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. A fast file system for UNIX. *ACM Transactions on Computer Systems 2*, 3 (August 1984), 181–197.

[6] PATTERSON, D., AND GIBSON, G. A case for redundant arrays of inexpensive disks (RAID). In *SIGMOD International Conference on Management of Data* (June 1988), pp. 109–116.

[7] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. In *ACM Transactions on Computer Systems* (Feb. 1992), vol. 10, pp. 26–52.

[8] SELTZER, M. I., BOSTIC, K., MCKUSICK, M. K., AND STAELIN, C. An implementation of a log-structured file system for UNIX. In *Proceedings of the 1993 Winter USENIX* (San Diego, CA, USA, January 1993), pp. 307–326.

[9] ZHU, N., CHEN, J., CHIUEH, T., AND ELLARD, D. Scalable and accurate trace replay for file server evaluation. Tech. Rep. TR153, SUNY Stony Brook, December 2004.

Aggressive Prefetching: An Idea Whose Time Has Come*

Athanasios E. Papathanasiou and Michael L. Scott
University of Rochester
{papathan,scott}@cs.rochester.edu
http://www.cs.rochester.edu/{~papathan,~scott}

Abstract

I/O prefetching serves to hide the latency of slow peripheral devices. Traditional OS-level prefetching strategies have tended to be conservative, fetching only those data that are very likely to be needed according to some simple heuristic, and only just in time for them to arrive before the first access. More aggressive policies, which might speculate more about which data to fetch, or fetch them earlier in time, have typically not been considered a prudent use of computational, memory, or bandwidth resources. We argue, however, that technological trends and emerging system design goals have dramatically reduced the potential costs and dramatically increased the potential benefits of highly aggressive prefetching policies. We propose that memory management be redesigned to embrace such policies.

1 Introduction

Prefetching, also known as prepaging or read-ahead, has been standard practice in operating systems for more than thirty years. It complements traditional caching policies, such as LRU, by hiding or reducing the latency of access to non-cached data. Its goal is to predict future data accesses and make data available in memory before they are requested.

A common debate about prefetching concerns how aggressive it should be. Prefetching aggressiveness may vary in terms of timing and data coverage. The timing aspect determines how early prefetching of a given block should occur. The data coverage aspect determines how speculative prefetching should be regarding which blocks are likely to be accessed. Conservative prefetching attempts to fetch data incrementally, just in time to be accessed, and only when confidence is high [3]. Aggressive prefetching is distinguished in two ways. First, it prefetches deeper in a reference stream, earlier than would be necessary simply to hide I/O latencies. Second, it speculates more about future accesses in an attempt to increase data coverage, possibly at the cost of prefetching unnecessary data.

The literature on prefetching is very rich (far too rich to include appropriate citations here). Researchers have suggested and experimented with history-based predictors, application disclosed hints, application-controlled prefetching, speculative execution, and data compression in order to improve prefetching accuracy and coverage for both inter- and intra-file accesses. Various methods have also been proposed to control the amount of memory dedicated to prefetching and the possible eviction of cached pages in favor of prefetching.

Published studies have shown that aggressive prefetching has the potential to improve I/O performance for a variety of workloads and computing environments, either by eliminating demand misses on pages that a conservative system would not prefetch, or by avoiding long delays when device response times are irregular. Most modern operating systems, however, still rely on variants of the standard, conservative sequential read-ahead policy. Linux, for example, despite its reputation for quick adoption of promising research ideas, prefetches only when sequential access is detected, and (by default) to a maximum of only 128 KB.

Conservative algorithms have historically been reasonable: aggressive prefetching can have a negative impact on performance. We begin by reviewing this downside in Section 2. In Section 3, however, we argue that the conventional wisdom no longer holds. Specifically, the risks posed by aggressive prefetching are substantially reduced on resource-rich modern systems. Moreover, new system design goals, such as power efficiency and disconnected or weakly-connected operation, demand the implementation of very aggressive policies that predict and prefetch data far ahead of their expected use.

*This work was supported in part by NSF grants EIA-0080124, CCR-0204344, and CNS-0411127; and by Sun Microsystems Laboratories.

Finally, in Section 4 we present new research challenges for prefetching algorithms and discuss the implications of those algorithms for OS design and implementation.

2 Traditional concerns

Under certain scenarios aggressive prefetching may have a severe negative impact on performance.

Buffer cache pollution. Prefetching deeper in a reference stream risks polluting the buffer cache with unnecessary data and ejecting useful data. This risk is particularly worrisome when memory is scarce, and when predictable (e.g. sequential) accesses to predictable (e.g. server disk) devices make it easy to compute a minimum necessary "lead time", and offer little benefit from working farther ahead.

Increased physical memory pressure. Aggressive prefetching may increase physical memory pressure, prolonging the execution of the page replacement daemon. In the worst case, correctly prefetched pages may be evicted before they have a chance to be accessed. The system may even thrash.

Inefficient use of I/O bandwidth. Aggressive prefetching requires speculation about the future reference stream, and may result in reading a large amount of unnecessary data. Performance may suffer if bandwidth is a bottleneck.

Increased device congestion. Aggressive prefetching leads to an increased number of *asynchronous* requests in I/O device queues. *Synchronous* requests, which have an immediate impact on performance, may be penalized by waiting for prefetches to complete.

Techniques exist to minimize the impact of these problems. More accurate prediction algorithms can minimize cache pollution and wasted I/O bandwidth. The ability to cancel pending prefetch operations may reduce the risk of thrashing if memory becomes too tight. Replacement algorithms that balance the cost of evicting an already cached page against the benefit of prefetching a speculated page can partially avoid the ejection of useful cached data (this assumes an on-line mechanism to accurately evaluate the effectiveness of caching and prefetching). Prefetched pages can be distinguished from other pages in the page cache so that, for example, they can use a different replacement policy (LRU is not suitable). Finally, priority-based disk queues that schedule requests based on some notion of criticality can reduce the impact of I/O congestion. All of these solutions, unfortunately, introduce significant implementation complexity, discouraging their adoption by general-purpose operating systems. In addition, most of the problems above are most severe on resource-limited systems. A general-purpose OS, which needs to run on a variety of machines, may forgo potential benefits at the high end in order to avoid more serious problems at the low end.

3 Why it makes sense now

Two groups of trends suggest a reevaluation of the conventional wisdom on aggressive prefetching. First, technological and market forces have led to dramatic improvements in processing power, storage capacity, and to a lesser extent I/O bandwidth, with only modest reductions in I/O latency. These trends serve to increase the need for aggressive prefetching while simultaneously decreasing its risks. Second, emerging design goals and usage patterns are increasing the need for I/O while making its timing less predictable; this, too, increases the value of aggressive prefetching.

3.1 Technological and market trends

Magnetic disk performance. Though disk latency has been improving at only about 10% per year, increases in rotational speed and recording density have allowed disk bandwidths to improve at about 40% per year [6]. Higher bandwidth disks allow—in fact require—the system to read more data on each disk context switch, in order to balance the time that the disk is actively reading or writing against the time spent seeking. In recognition of this fact, modern disks have large controller caches devoted to speculatively reading whole tracks. In the absence of memory constraints, aggressive prefetching serves to exploit large on-disk caches, improving utilization.

Large memory size at low cost. Memory production is increasing at a rate of 70% annually, while prices have been dropping by 32% per year [5], reaching a cost today of about 12 cents per megabyte. Laptop computers with 512 MB of memory are now common, while desktop and high-end systems may boast several Gigabytes. While application needs have also grown, it seems fair to say on the whole that today's machines have significantly more memory "slack" than their predecessors did, providing the opportunity to prefetch aggressively with low risk of pollution or memory pressure.

Figure 1 presents laptop memory availability and company recommended memory sizes for commercial operating systems and applications since 1995. Memory availability is shown using three possible configurations: the *maximum* and *minimum* available, and a "*low cost*" option obtained by filling every memory slot with the density of RAM that maximized MB/dollar in the technology of the day. A comparison between the recommended memory size for our most memory-intensive application, Photoshop, and the low-cost memory configuration shows that the available memory "slack" in a low-cost laptop has grown from less than 1 MB in 1995 to

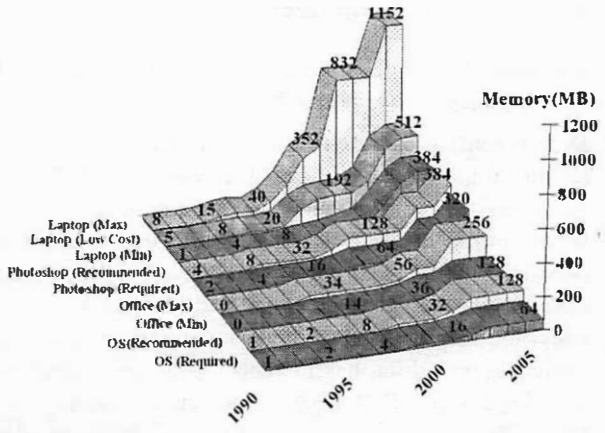


Figure 1: Memory "slack" trends. The figure presents how laptop memory availability and memory requirements of commercial applications and operating systems have changed during the last 15 years. Memory availability data are based on Macintosh laptop systems [1] and are represented using three possible configurations: the *maximum* and *minimum* available, and a "*low cost*" option obtained by filling every memory slot with the density of RAM that maximized MB/dollar in the technology of the day. Memory requirement data are based on company recommendations for Microsoft Windows operating systems and two applications, Adobe Photoshop and Microsoft Office. For Microsoft Windows and Adobe Photoshop company *recommended* and *required* memory sizes are provided. For Microsoft Office, *Min* represents the memory required to run a single office application and *Max* represents the memory required to run six office applications concurrently.

more than 100MB today. The available "slack" is significantly higher for high-end laptops and desktops. As memory sizes and disk bandwidths continue to increase, and as multimedia applications continue to proliferate (more on this below), the performance benefit of aggressive prefetching will surpass that of caching policies.

I/O performance variability. In the past, prefetching algorithms have been developed under the assumption that storage devices are able to deliver data at relatively constant latencies and bandwidths. This assumption no longer holds. First, users access data through multiple devices with different performance characteristics. Second, the performance of a given device can vary greatly. Increases in areal densities of magnetic disks have led to bandwidth differences of 60% or more [7] between inner and outer tracks, and this gap is expected to grow. Li *et al.* [10] demonstrate a 15% to 47% throughput improvement for server-class applications through a competitive prefetching algorithm that takes into account the performance variability of magnetic disks. Similarly, wireless network channels are prone to noise and contention, resulting in dramatic variation in bandwidth over time. Finally, power-saving modes in disks and other devices can lead, often unpredictably, to very large increases in latency. To maintain good performance and make efficient use of the available bandwidth, prefetching has to be sensitive to a device's performance variation. Aggressive prefetching serves to exploit periods of higher bandwidth, hide periods of higher latency, and generally smooth out fluctuations in performance.

The processor–I/O gap. Processor speeds have been doubling every 18 to 24 months, increasing the performance gap between processors and I/O systems. Processing power is in fact so far ahead of disk latencies that prefetching has to work multiple blocks ahead to keep the processor supplied with data. Moreover, many applications exhibit phases that alternate between compute-intensive and I/O-intensive behavior. To maximize processor utilization, prefetch operations must start early enough to hide the latency of the *last* access of an I/O-intensive phase. Prefetching just in time to minimize the impact of the *next* access is not enough. Early prefetching, of course, implies reduced knowledge about future accesses, and requires both more sophisticated and more speculative predictors, to maximize data coverage. Fortunately, modern machines have sufficient spare cycles to support more computationally demanding predictors than anyone has yet proposed. In recognition of the increased portion of time spent on I/O during system start-up, Microsoft Windows XP employs history-based informed prefetching to reduce operating system boot time and application launch time [12]. Similar approaches are being considered by Linux developers [8].

3.2 Design goals and usage patterns

Larger data sizes. Worldwide production of magnetic content increased at an annual rate of 22% from 1999 to 2000 [11]. This increase has been facilitated by annual increases in disk capacity of 130% [19]. Multimedia content (sound, photographs, video) contributes significantly to and will probably increase the growth rate of on-line data. As previous studies have shown [2, 14], larger data sets diminish the impact of larger memories on cache hit rates, increasing the importance of prefetching. Media applications also tend to touch each datum only once, limiting the potential of caching.

Multitasking. The dominance and maturity of multitasking computer systems allow end users to work concurrently on multiple tasks. On a typical research desktop, it is easy to imagine listening to a favorite MP3 track while browsing the web, downloading a movie, rebuilding a system in the background, and keeping half an eye on several different instant messaging windows. The

constant switching among user applications, several of which may be accessing large amounts of data, reduces the efficiency of LRU-style caching; aggressive prefetching allows each application to perform larger, less frequent I/O transfers, exploiting the disk performance advances described above.

Energy and power efficiency. Energy and power have become major issues for both personal computers and servers. Components of modern systems—and I/O devices in particular—have multiple power modes. Bursty, speculative prefetching can lead to dramatic energy savings by increasing the time spent in non-operational low-power modes [15]. As shown in our previous work [16], we can under reasonable assumptions[1] save as much as 72% of total disk energy even if only 20% of what we prefetch actually turns out to be useful. At the same time, *operational* low-power modes, as in recent proposals for multi-speed disks [4], suggest the need for *smooth* access patterns that can tolerate lower bandwidth. Prefetching algorithms for the next generation of disks may need to switch dynamically between smooth low-bandwidth operation and bursty high-bandwidth operation, depending on the offered workload.

Reliability and availability of data. Mobile systems must increasingly accommodate disconnected or weakly-connected operation [9], and provide efficient support for several portable storage devices [13]. Reliability and availability have traditionally been topics of file system research. We believe, however, that memory management and specifically prefetching must play a larger role. Mobile users may need to depend on multiple legacy file systems, not all of which may handle disconnection well. But while computers may have multiple file systems, they have a *single* memory management system. The file system functionality that provides access under disconnected or weakly-connected operation is based on aggressive, very speculative prefetching (with caching on local disk). This prefetching can be moved from the low-level file system to the memory management and virtual file system layers, where it can be used in conjunction with arbitrary underlying file systems. Aggressive prefetching that monitors access patterns through the virtual file system and is implemented at the memory management level might prefetch and back up data to both RAM and local peripheral devices.

[1] We assume 50 MB of memory dedicated to prefetching and an application data consumption rate of 240 KB/s (equivalent to MPEG playback). Energy savings are significant for several combinations of memory sizes used for prefetching, data rates and prefetching accuracy ratings.

4 Research challenges

The trends described in Section 3 raise new design challenges for aggressive prefetching.

Device-centric prefetching. Traditionally, prefetching has been application-centric. Previous work [17] suggests a cost-benefit model based on a constant disk latency in order to control prefetching. Such an assumption does not hold in modern systems. To accommodate power efficiency, reliability, and availability of data under the varying performance characteristics of storage devices, prefetching has to reflect both the application *and* the device. Performance, power, availability, and reliability characteristics of devices must be exposed to prefetching algorithms [10, 13, 16].

Characterization of I/O demands. Revealing device characteristics is not enough. To make informed decisions the prefetching and memory management system will also require high level information on access patterns and other application characteristics. An understanding of application reliability requirements, bandwidth demands, and latency resilience can improve prefetching decisions.

Coordination. Non-operational low-power modes depend on long idle periods in order to save energy. Uncoordinated I/O activity generated by multitasking workloads reduces periods of inactivity and frustrates the goal of power efficiency. Aggressive, coordinated prefetching can be used in order to coordinate I/O requests across multiple concurrently running applications and several storage devices.

Speculative predictors that provide increased data coverage. Emerging design goals, described in Section 3, make the case to prefetch significantly deeper than traditional just-in-time policies would suggest. In addition to short-term future data accesses, prefetching must predict long-term user intention and tasks in order to minimize the potentially significant energy costs of misses (e.g. for disk spin-up) and to avoid the possibility of application failure during weakly-connected operation.

Prefetching and caching metrics. Traditionally, cache miss ratios have been used in order to evaluate the efficiency of prefetching and caching algorithms. The utility of this metric, however, depends on the assumption that all cache misses are equivalent [18]. Power efficiency, availability, and varying performance characteristics lead to different costs for each miss. For example, a miss on a spun-down disk can be significantly more expensive in terms of both power and performance than a miss on remote data accessed through the network. We need new methods to evaluate the effectiveness of proposed prefetching algorithms.

In addition, several traditional prefetching problems may require new or improved solutions as prefetching becomes more aggressive. Examples include:

- Separate handling of prefetched and cached, accessed pages.
- Algorithms that dynamically control the amount of physical memory dedicated to prefetching.
- Monitoring systems that evaluate the efficiency of predictors and prefetching algorithms using multiple metrics (describing performance, power efficiency, and availability) and take corrective actions if necessary.
- Priority-based disk queues that minimize the possible negative impact of I/O queue congestion.
- Mechanisms to cancel in-progress prefetch operations in the event of mispredictions or sudden increases in memory pressure.
- Data compression or other techniques to increase data coverage.

To first approximation, the memory management system of today assumes responsibility for caching secondary storage without regard to the nature of either the applications above it or the devices beneath it. We believe this has to change. The "storage management system" of the future will track and predict the behavior of applications, and prioritize and coordinate their likely I/O needs. At the same time, it will model the latency, bandwidth, and reliability of devices over time, moving data not only between memory and I/O devices, but among those devices as well, to meet user-specified needs for energy efficiency, availability, reliability, and interactive responsiveness.

References

[1] Apple Computer, Inc. Apple Specifications. Available: http://www.info.apple.com/support/applespec.html.

[2] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a Distributed File System. In *Proc. of the 13th ACM Symp. on Operating Systems Principles*, 1991.

[3] P. Cao, E. W. Felten, and K. Li. A Study of Integrated Prefetching and Caching Strategies. In *Proc. of the 1995 ACM Joint Int. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'95/PERFORMANCE'95)*, 1995.

[4] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. DRPM: Dynamic Speed Control for Power Management in Server Class Disks. In *Proc. of the 30th Int. Symp. on Computer Architecture (ISCA'03)*, June 2003.

[5] J. Handy. Will the Memory Market EVER Recover?, Sept. 1999. http://www.reed-electronics.com/electronicnews/article/CA47864.html.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 3rd edition, 2003.

[7] Hitachi Global Storage Technologies. Hard Disk Drive Specification: Hitachi Travelstar 5K80 2.5 inch ATA/IDE hard disk drive, Nov. 2003.

[8] Kerneltrap.org. Linux: Boot Time Speedups Through Precaching, Jan. 2004. http://kerneltrap.org/node/2157.

[9] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Trans. on Computer Systems*, 10(1), Feb. 1992.

[10] C. Li, A. E. Papathanasiou, and K. Shen. Competitive Prefetching for Data-Intensive Online Servers. In *Proc. of 1st Workshop on Operating Sys. and Arch. Support for the on-demand IT InfraStructure*, Oct. 2004.

[11] P. Lyman and H. R. Varian. How Much Information, 2003. School of Information Management and Systems, University of California at Berkeley. Available: http://www.sims.berkeley.edu/how-much-info-2003/.

[12] Microsoft Corporation. Kernel Enhancements for Windows XP, Jan. 2003. http://www.microsoft.com/whdc/driver/kernel/XP_kernel.mspx.

[13] E. B. Nightingale and J. Flinn. Energy-Efficiency and Storage Flexibility in the Blue File System. In *Proc. of the 6th USENIX Symp. on Operating Systems Design and Implementation*, Dec. 2004.

[14] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A Trace-driven Analysis of the UNIX 4.2 BSD File System. In *Proc. of the 10th ACM Symp. on Operating Systems Principles*, Dec. 1985.

[15] A. E. Papathanasiou and M. L. Scott. Energy Efficiency Through Burstiness. In *Proc. of the 5th IEEE Workshop on Mobile Computing Systems and Applications*, Oct. 2003.

[16] A. E. Papathanasiou and M. L. Scott. Energy Efficient Prefetching and Caching. In *Proc. of the USENIX 2004 Annual Technical Conf.*, June 2004.

[17] R. H. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, Dec. 1995.

[18] M. Satyanarayanan. Mobile Computing: Where's the Tofu? *ACM SIGMOBILE Mobile Comp. and Comm. Review*, 1(1), Apr. 1997.

[19] G. Zeytinci. Evolution of the Major Computer Storage Devices, 2001. Available: http://www.computinghistorymuseum.org/teaching/papers/research/StorageDevices-Zeytinci.pdf.

Why Markets Could (But Don't Currently) Solve Resource Allocation Problems in Systems

Jeffrey Shneidman[1], Chaki Ng[1], David C. Parkes[1]
Alvin AuYoung[2], Alex C. Snoeren[2], Amin Vahdat[2], and Brent Chun[3]
[1]*Harvard University,* [2]*University of California, San Diego,* [3]*Intel Research, Berkeley*

Abstract

Using market mechanisms for resource allocation in distributed systems is not a new idea, nor is it one that has caught on in practice or with a large body of computer science research. Yet, projects that use markets for distributed resource allocation recur every few years [1, 2, 3], and a new generation of research is exploring market-based resource allocation mechanisms [4, 5, 6, 7, 8] for distributed environments such as Planetlab, Netbed, and computational grids.

This paper has three goals. The first goal is to explore why markets can be appropriate to use for allocation, when simpler allocation mechanisms exist. The second goal is to demonstrate why a new look at markets for allocation could be timely, and not a re-hash of previous research. The third goal is to point out some of the thorny problems inherent in market deployment and to suggest action items both for market designers and for the greater research community. We are optimistic about the power of market design, but we also believe that key challenges exist for a markets/systems integration that must be overcome for market-based computer resource allocation systems to succeed.

1 Is there a Problem?

During the past decade, we have witnessed the emergence of systems that are owned, deployed, and used by multiple self-interested stakeholders. Consider the differences between traditional distributed systems and current distributed environments, such as Planetlab, Netbed, and computational grids. Current environments have the following properties:

Many resources, many users, and more complicated needs. Multiple self-interested parties can simultaneously supply and consume sets of resources (e.g., machine time, bandwidth). Users can demand large sets of disparately controlled resources, creating a large combinatorial allocation problem not easily solved by techniques like social pairwise agreements.

Resource demand exceeds resource supply. Previous work has graphically demonstrated this problem on Planetlab, where the machine load is many times the system capacity [9]. Scientific computing (grid) users expect this to be a problem as they deploy experimental testbeds [10].

No job selection by committee. The scale and design goals of these systems preclude an administrative body to handle resource allocation.

Incentives and external constraints limit supply. Political, financial, and geographic limitations prevent additional hardware deployments to solve all cases of resource contention. Unlike commercial servers that have a financial incentive to support their peak user load, resource providers in shared environments usually have little incentive to add resources to the shared system.

Testbed-sensitive experimentation. In some shared environments (e.g., Planetlab), the network itself is the target of research. A *tragedy of the commons* [11] can develop where overlapping usage consumes resources to the point of disutility and users are unable to run certain class of measurement experiments accurately or at all.

Computer systems have reached the point where the goal of distributed resource allocation is no longer to maximize utilization; instead, when demand exceeds supply and not all needs can be met, one needs a policy for making resource allocation decisions. Researchers (Planetlab central, Grid planners, etc.) have

started to consider more intelligent ways of allocating resources than simple best effort, or randomized allocation schemes.

These methods can involve a *social policy* for resource distribution. A policy is simply a set of rules for allocation when resource demand exceeds resource supply. One candidate policy is to seek *efficient* usage, which directs a mechanism to allocate resources to the set of users who have the highest utility for the use of the resources. Other social policies exist, such as those that favor small experiments, or favor underrepresented stakeholders, or (if money is involved) seek maximal revenue generation. One can also implement a mixture of policies to meet a complex social goal.

Past deployments of distributed system schedulers (e.g. Condor [12]) focused on maximizing utilization, and were not designed to support complex social policy. Today's schedulers must take full utilization as the common case and focus on solving the resulting resource contention problems.

In this paper, we explore the idea of using market-based mechanisms to address resource allocation problems in distributed systems. In Sections 2 and 3 we explore how markets may be a useful (and perhaps required) tool in this research and why they warrant new consideration by systems researchers. However, there are special challenges that arise when markets are used for computational resource allocation. These challenges, presented in Section 4, could prove overwhelming depending on the response from the systems community and our collective ability to address these concerns.

We feel that now is a critical time for the systems community to consider the various resource allocation capabilities that should be supported in next-generation distributed systems, before an uninformed decision or simple necessity leads to a less desirable, de facto standard.

2 The Role of Markets

If one is interested in performing policy-directed resource allocation, one should consider allocation schemes that are based around a market.

A market is a way for buyers and sellers to exchange goods. Applied to computer resource allocation, the traded goods could be the right to use a certain amount of system resources on a set of machines. When demand exceeds supply, markets provide a goal-oriented way of allocating resources among competing interests while meeting some social goal. One natural goal is to maximize overall "happiness" or utility of the users. When users have complex needs, achieving this goal is not easy for either the individual users and for the system tasked with making the allocation decision. We will return to these issues in Section 4, but for now we consider the advantages of markets for computational resource allocation.

Deploying a computational market for resource allocation in the systems domain can benefit two research constituencies. The first constituency, which will be ignored for the rest of this paper, are the experimental economists and economically-minded computer scientists. Rarely are economists actually given the opportunity to deploy a market or a whole economy, let alone several for comparison. Computational mechanism design [13] is an emerging topic partly because the results apply to many different domains, and there is some merit in asking systems researchers to be research subjects as they attempt to use some market mechanism for their own work.

But systems researchers (the second constituency) are much more interested in knowing if these proposed market allocation projects and their system offspring solve real problems in distributed resource allocation. There are many programmatic alternatives to markets in resource allocation. These include simple first come-first served allocation, reservation systems, and more elaborate systems such as automated voting schemes or other devices. Unlike these simpler ideas, market-based systems can naturally address the new-world system characteristics described in Section 1. Namely, market-based systems can:

Provide a "socially optimal" project director to resolve overdemand. Unlike simpler mechanisms, markets can support a rich set of social goals, such as finding an efficient allocation decision. The most natural way to reach an efficient decision is to require users to quantify their perceived benefit of winning their resource request. A market encourages participants to use resources wisely and tries to make an overall usage decision to maximize overall value.

Provide incentives for growth. Markets are often used along with a currency that can be used to express value and acts as a medium of exchange.[1] If a currency is *open* and can be used to acquire a multitude of goods and services, then this currency can be used to incent resource providers to expand their services. In contrast, a *closed* currency can incent growth only if the receiver of the currency has some use for its receipt. One can use currency to create a medium to allow a market's "invisible hand"

[1] Currency is a natural means toward easy valuation expression, but there are other allocation algorithms that do not require currency. An example are the matching algorithms that link Medical Interns and Residents in the United States [14]. In this setting, medical students and residency programs bid on each other using a prioritization scheme, and these bids are resolved with a winner determination algorithm. At first blush, a matching market does not seem appropriate to systems resource allocation problems, where sellers have no preference of who uses their resources.

to reward those who provide useful resources to the network. Markets provide a vetted set of payment rules that can be used to transfer currency between buyers and sellers.

Provide a vocabulary to describe complex resource bundles. In any system, be it administrative or market-based, users need a mechanism to express their resource holdings and desires. Markets, which have been used for decades to capture difficult resource allocation problems (e.g. energy markets, wireless spectrum auctions, airline landing slot exchanges), can also be used to capture the intricacies of systems problems. Bidding languages have been studied for their tradeoffs between expressivity and compactness [15], and existing languages can be directly applied to computer resources.

Link Cross-Testbed experimentation. Multiple closed distributed systems that run in parallel can offer unique resources such as access to specific scientific equipment. One can imagine a physics researcher willing to provide access to their Beowulf cluster [16] but wishing to consume resources produced by data collectors at a CERN [17] on a completely separate network. Linked market-based mechanisms could be used to quantify the value of the cluster time sold in one network and the value of a CERN resource purchased in another network in a manner similar to how real economies are linked through a a currency exchange. Ongoing research into exchange mechanisms for computational systems could make this vision feasible [7].

3 Not Déjà Vu All Over Again

The idea of using markets and pricing computer resources is quite old. Pricing policies received considerable attention at the dawn of modern multi-user time sharing systems. Papers in the late 1960's were dedicated to automated pricing policies for computer time [18, 19, 20]. As research, this work was short-lived. The complexity of these schemes relative to their benefit, combined with the environment of time-shared systems (mostly cooperative, mostly controlled by a single entity) quickly made pricing for shared resource allocation a low priority. Shared resource allocation remained a hot topic in operating systems, but the goal in this research was maximizing *utilization* through clever scheduling. In contrast, schedulers that promote social goals such as *efficient* usage have not been as widely investigated.

This said, there have been past systems that take a market approach to resource allocation [1, 2]. How, then, will new research into markets for distributed resource allocation be any different? We believe that a number of developments make the timing right to revisit the question of whether market-based models are both appropriate and, more importantly, required for emerging computational environments. New research can take advantage of the following developments:

Pressing demand. Past market-based systems never saw real field testing, and contention was often artificially generated. Today, a deployed market system could have immediate usage and solve real resource conflicts. Real usage data will help researchers calibrate and evaluate their market-based resource schedulers. Previous mechanism designs were not able to take advantage of user feedback to drive the mechanism design process.

Improved operating system infrastructure. Past systems had to deal with limitations in infrastructure, such as a lack of user authentication or kernel-supported resource isolation. Today, systems research has produced tools like BSD Jails, Xen, and Linux CKRM [21, 22], which are already in use to provide resource isolation, can be adopted to enforce allocation decisions.

Expressive market design. Previous work used bidding languages that have been artificially limited in their expressive power. During the past decade, tremendous advances have been made in the theory and practice of expressive market design. Current mechanisms can support combinatorial bidding, which more naturally captures resource needs. For instance, modern bidding languages can easily represent any logical combination of goods, such as AND, OR, XOR, and CHOOSE. This expressive power did not exist in previous mechanism deployments.

Scalable mechanisms. Solving large resource contention problems has traditionally been computationally expensive. Fortunately, significant advances have been made in the theory of solving large-scale mixed-integer optimization problems, which is an underlying technology well-suited to implementing market problems. This theory is now reflected in off-the-shelf solvers such as CPLEX. Significant breakthroughs have arisen from the use of cutting plane techniques, branch-and-cut, and preprocessing to achieve efficient solving.

4 Markets/Systems Integration Challenges

Despite our general optimism, the ultimate success of a deployed mechanism is measured in usage, and usage depends on a number of factors typically overlooked by computer science researchers. Ease of use may trump mechanism features. People may be willing to accept

the limitations of simpler systems (eg: first-come first-served, or randomized allocation) if market-based systems are seen as too complex, or if they fail in other ways, even if accepting a simpler system means ignoring some of the characteristics described in Section 1.

In this section, we articulate the roadblocks that must be addressed to make a market/systems integration successful. In our opinion, these challenges are not in the market details. Rather, we think that the biggest challenges to their adoption in systems will come from understanding, supporting, and using these mechanisms. After presenting each challenge, we consider *action items* for the general systems community, as well as for systems market designers where appropriate. In our view, a markets/systems integration could fail if these challenges are not overcome:

Allocation Policy Must be Explicit. One of the uncomfortable realities of a market is that it forces user communities to confront their social allocation rules. Do people want allocative efficiency? Do people want testbeds to be self-sustaining through policies that imply taxation? Do people want to favor jobs from underrepresented users? Other real-world uses of markets have had definite mandates. As an example, after years of running a lottery to allocate wireless spectrum, the U.S. Congress wised up to the resulting allocation inefficiency (not to mention the possibilities of revenue generation with the government as the initial sole seller), and mandated that the F.C.C. to employ an efficient allocation mechanism. This was a clear social choice, and necessarily meant the F.C.C. used a market.

Community Action Items: There is no general mandate in the systems community for the social goal of an allocation scheme. If the systems community cares about simpler goals than efficiency or revenue generation, than systems market designers should not be trying to develop auction mechanisms. Where should this mandate come from? HotOS participants? Planetlab Central? Grid users?

Dividing Up Resources as a Seller. Unlike many other markets, there are complex and not commonly understood systems interactions between computer resources, complicating the allocation decision. Consider a system that allocates three hard resources, CPU, memory, and disk: An allocation of memory is meaningless unless there is some small CPU associated with the allocation. If virtual memory is involved, it is likely that disk also needs to be allocated, but that the effects of swapping will dominate the time required to run the experiment. Either these associations are explicit, in which case minimum resource bundles must be purchased, or there are side effects that constrain the allocation based on the characteristic of winning bids.

Systems Market Designer Action Items: While the tools (like CKRM [22]) for partitioning resources are being developed, they still have a long way to go to capture pertinent resources and even trivial resource interactions.

Predicting Needs as a Buyer. It is difficult to describe precisely the level of resources required to run an experiment or job. Depending on the inputs to a program, the ideal level of resource consumption can vary dramatically.

Moreover, there is a tangible penalty for misestimating resource need, since these bids are made in advance of when the resources will actually be available. In order to match enough buyers with sellers, current market-based resource allocation schemes batch allocations into blocks of time. The time scale of this batch system can be minutes or days ahead of when the resources will actually be made available. This means that users must predict their resource needs in advance. A resource underbid will prove unsatisfying if won, while a resource overbid (with the same value) is less likely to win because of competition from more efficient users. Requiring users to predict their resource need is new user behavior, and this forecasting problem can be difficult.

Community Action Items: The general systems community should think more about building tools to help users estimate their resource needs. Perhaps users in a shared environment will have access to a best-effort staging ground where they will be able to gauge their resource usage. One can imagine future research tools (either modeling or analysis) that attempt to capture the resource profile of a wide-area application. Such tools are an open area for ongoing and future research [10]. *Systems Market Designer Action Items:* While there is ongoing research into online market mechanisms—making an allocation decision before seeing all bid activity—designers should develop markets that are less rigid in their clearing time frames, while still meeting social goals.

Valuing Resources. Utility maximizing market mechanisms are only as accurate as the values that users assign their bids (on goods that they possess, and goods that they would like to acquire). But what is a user's true value on four hours of CPU time, a week before a major conference deadline? (*Any* situation where demand exceeds supply will lead to unhappy users; a variation of this question exists in any resource allocation scenario.) Ultimately, the requirement of the market is that users place a value on their resource needs and holdings. There are several problems with calculating this value in computational systems. We label these as problems with a

well-defined currency, and in calculating and expressing valuation:

Well-Defined Currency. Almost all previously deployed computational markets have used a virtual currencies instead of real cash. The low barrier to utilization and low stakes in case of deployment error make simple closed virtual currencies attractive to developers. In these scenarios, it is all too easy to skip the monetary policy considerations that make currencies work.

For all of their bootstrapping advantages, virtual currencies require initial thought and ongoing care to function properly. Virtual currencies often suffer from a lack of liquidity, making it difficult to convert into or out of the virtual currency. As a result, these ersatz currencies are quite limited; certain users might be willing to sell resources for Euros, but not for un-exchangable Woozies. Furthermore, virtual currencies can suffer from *starvation*, as heavy consumers run out of currency to spend, *depletion* as users leave the system or hoard currency reducing the total amount of currency available to others, and *inflation* as users are added to the system with an initial credit. Previous research attempts to address the faults of virtual currency systems with monetary policies and administrative measures (e.g., [23]), but for a virtual currency to work, it must be expressive and appreciated by users.[2]

We believe that the success of a computational resource exchange will be tied to a well-defined currency. Rather than attempting to create such a currency, one could turn to real money as the medium for exchange. One reason to use a real currency is that it may increase resource contribution and ease maintenance of distributed environments. Using Netbed or Planetlab as an example, many entities are passive, light users, and may not see the value of maintaining their portion of the network beyond their initial required contribution. Whereas these users may not respond to an allocation of a closed virtual currency, they may respond to real money. Using a real currency could help increase participation in a distributed system – since supply and demand set the price of contributed resources, the network has a way of rewarding those who provide useful offerings to the network. Using a real currency also might provide a lower barrier to entry for new users and create a self-sustaining shared environment: rather than charging new organizations a fixed usage fee, or relying on external grant money for support, one can imagine transaction fees that support the development of the testbed.

We believe that there is no technical reason that prevents one from using real currencies on shared environments. There are numerous political and fairness concerns with this idea. Researchers don't like the idea of having a resource request denied because other researchers could pay more money. (We do observe that the existing research grant process potentially creates this sort of situation.) But in a world where demand exceeds supply, and one has chosen to resolve this problem efficiently, one needs some understood way of expressing valuation differences. Perhaps using a real currency is a wacky idea (that works for every other market) whose time has come?

Community Action Items: If efficiency is an important social goal, then we see valuation questions as a big challenge for the systems community. We wonder if users would be willing to try something novel (which is old hat to every other use of markets) and pay for their bids with real currency. While there are issues with this idea, it does force people to put money where their valuations are. *Systems Market Designer Action Items:* We would like to see a careful construction of a virtual currency system, or alternatively, a careful construction of an argument as to why these systems do not work. We feel that a well-defined currency is a major stumbling block to market adoption in systems.

Calculating and Expressing Valuation. It can be difficult for a user to accurately value their ideal resource bundles. There needs to be a simple and effective way for people to express their resource need and calculate its value. To stress this point, imagine a market interface that asked the user for their valuation, one question at a time, over the entire space of good combinations. This painful approach would require the user to think about their valuation for a whole slew of bundles, a time-consuming and sometimes difficult task. An area of market design that has received almost no attention for computer resources is in the user interface between the users and the mechanism. The bidding interface is the most public face of a market mechanism, and in our opinion it is this interface that has the greatest effect on user perception (and acceptance) of the mechanism as a useful tool.

Community Action Items: Be willing to give feedback to designers on how well a language/interface is at capturing your resource desires. Be willing to suffer through some bad research designs. *Systems Market Designer Action Items:* Improving price guidance and addressing

[2] One interesting note is that the new breed of multi-player online games often have a virtual in-game currency component. Operators of these online games either openly support the exchange of their currency into other real currencies [24], or attempt to keep their currency closed, effectively incenting players to open these closed currencies by spawning parallel side exchange markets [25].

valuation complexity are currently active research areas in mechanism design, and this effort will likely continue.

5 Conclusion and Challenges

We feel that the time is right to explore market-based resource allocation mechanisms, but we also see a number of challenges that may hinder their applicability to systems. While there has been a general call for better resource allocation, it is not clear to us that systems researchers will be willing to accept the implications of mechanisms to achieve certain social goals. These market designs need to be debated, and if deemed valuable, deployed and evaluated "in the wild".

References

[1] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and S. Stornetta, "Spawn: A distributed computational economy," *IEEE Transactions on Software Engineering*, vol. 18, no. 2, pp. 103–177, February 1992.

[2] A. S. Tanenbaum, S. J. Mullender, and R. van Renesse, "Using sparse capabilities in a distributed operating system," in *Proceedings of the 6th International Conference on Distributed Computing Systems*, 1986, pp. 558–563.

[3] O. Regev and N. Nisan, "The popcorn market – an online market for computational resources," in *Proceedings of the 1st International Conference on Information and Computation Economies*, October 1998.

[4] R. Wolski, J. Plank, and J. Brevik, "G-commerce – building computational marketplaces for the computational grid," The University of Tennessee at Knoxville, Tech. Rep. CS-00-439, Apr. 2000.

[5] R. Buyya, J. Giddy, and D. Abramson, "A case for economy grid architecture for service-oriented grid computing," in *Proc. of HCW '01*, Apr. 2001.

[6] A. AuYoung, B. N. Chun, A. C. Snoeren, and A. Vahdat, "Resource allocation in federated distributed computing infrastructures," in *Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-demand IT InfraStructure*, October 2004.

[7] D. Parkes, R. Cavallo, N. Elprin, A. Juda, S. Lahaie, B. Lubin, L. Michael, J. Shneidman, and H. Sultan, "ICE: An iterative combinatorial exchange," in *Proc. of ACM Conference on Electronic Commerce*, June 2005.

[8] B. A. H. Kevin Lai and L. Fine, "Tycoon: A Distributed Market-based Resource Allocation System," HP Labs, Palo Alto, CA, USA, Tech. Rep. arXiv:cs.DC/0404013, Apr. 2004.

[9] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat, "SHARP: An architecture for secure resource peering," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[10] S. Grinstein, J. Huth, and J. Schopf, "Resource predictors in hep applications," in *Proc. of Computing in High Energy and Nuclear Physics (CHEP)*, September 2004.

[11] G. Hardin, "The tragedy of the commons," in *Science*, 162(1968):1243-1248.

[12] "The Condor Project: http://www.cs.wisc.edu/condor/."

[13] N. Nisan and A. Ronen, "Algorithmic mechanism design (extended abstract)," in *STOC '99: Proceedings of the thirty-first annual ACM symposium on Theory of computing*. New York, NY, USA: ACM Press, 1999, pp. 129–140.

[14] A. E. Roth, "Game Theory as a Tool for Market Design," http://kuznets.fas.harvard.edu/ aroth/design.pdf.

[15] N. Nisan, "Bidding and allocation in combinatorial auctions," in *Proceedings of the 2nd ACM Conference on Electronic Commerce*, October 2000.

[16] "Beowulf Cluster Project: http://www.beowulf.org/."

[17] "CERN Particle Physics Experiments: http://www.cern.ch/."

[18] D. S. Diamond and L. L. Selwyn, "Considerations for computer utility pricing policies," in *Proceedings of the 1968 23rd ACM national conference*. ACM Press, 1968, pp. 189–200.

[19] I. E. Sutherland, "A futures market in computer time," *Commun. ACM*, vol. 11, no. 6, pp. 449–451, 1968.

[20] F. J. Corbato and V. A. Vyssotsky, "Introduction and overview of the multics system," http://www.multicians.org/fjcc1.html.

[21] P. R. Barham, B. Dragovic, K. A. Fraser, S. M. Hand, T. L. Harris, A. C. Ho, E. Kotsovinos, A. V. Madhavapeddy, R. Neugebauer, I. A. Pratt, and A. K. Warfield, "Xen 2002," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-553, Jan. 2003. [Online]. Available: http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-553.pdf

[22] "Class-based Kernel Resource Management (CKRM): http://ckrm.sourceforge.net/."

[23] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer, "Karma: A secure economic framework for p2p resource sharing," in *Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, 2003.

[24] "Online Gaming gets Feeding Tube," http://wired-vig.wired.com/news/ebiz/0,1272,67277,00.html.

[25] "Gaming Open Market," http://www.gamingopenmarket.com/.

Operating Systems Should Support Business Change

Jeffrey C. Mogul
HP Labs, Palo Alto
Jeff.Mogul@hp.com

Abstract

Existing enterprise information technology (IT) systems often inhibit business flexibility, sometimes with dire consequences. In this position paper, I argue that operating system research should be measured, among other things, against our ability to improve the speed at which businesses can change. I describe some of the ways in which businesses need to change rapidly, speculate about why existing IT infrastructures inhibit useful change, and suggest some relevant OS research problems.

1 Introduction

Businesses change. They merge; they split apart; they reorganize. They launch new products and services, retire old ones, and modify existing ones to meet changes in demand or competition or regulations. Agile businesses are more likely to thrive than businesses that cannot change quickly.

A business can lack agility for many reasons, but one common problem (and one that concerns us as computer scientists) is the inflexibility of its IT systems. "Every business decision generates an IT event" [1]; For example, a decision to restrict a Web site with product documentation to customers with paid-up warranties requires a linkage between that Web site and the warranty database. If the IT infrastructure deals with such "events" slowly, the business as a whole will respond slowly; worse, business-level decisions will stall due to uncertainty about IT consequences.

What does this have to do with operating systems? Surely the bulk of business-change problems must be resolved at or above the application level, but many aspects of operating system research are directly relevant to the significant problems of business change. (I assume a broad definition of "operating system" research that encompasses the entire, distributed operating environment.)

Of course, support for change is just one of many problems faced by IT organizations (ITOs), but this paper focusses on business change because it seems underappreciated by the systems software research community. We are much better at problems of performance, scale, reliability, availability, and (perhaps) security.

2 IT vs. business flexibility

Inflexible IT systems inhibit necessary business changes. The failure to rapidly complete an IT upgrade can effectively destroy the value of a major corporation (e.g., [12]). There is speculation that the Sept. 11, 2001 attacks might have been prevented if the FBI had had more flexible IT systems [17, page 77]. Even when IT inflexibility does not contribute to major disasters, it frequently imposes costs of hundreds of millions of dollars (e.g., [13, 14]).

The problem is not limited to for-profit businesses; other large organizations have similar linkages between IT and their needs for change. For example, the military is a major IT consumer with rapidly evolving roles; hospitals are subject to new requirements (e.g., HIPAA; infection tracking); universities innovate with IT (e.g., MIT's OpenCourseWare); even charities must evolve their IT (e.g., for tracking requirements imposed by the USA PATRIOT Act). The common factor is a large organization that thinks in terms of buying "enterprise IT" systems and services, not just desktops and servers.

3 Why is application deployment so slow?

IT organizations often spend considerably more money on "software lifecycle" costs than on hardware purchases. These costs include software development, testing, deployment, and maintenance. In 2004, 8.1% of worldwide IT spending went to server and storage hardware combined, 20.7% went to packaged software, but 41.6% went to "services," including 15.4% for "implementation" [15]. Even after purchasing packaged software, IT departments spend tons of money actually making it work [12].

Testing and deployment also impose direct hardware costs; for example, roughly a third of HP's internal servers are dedicated to these functions, and the fraction is larger at some other companies [21]. These costs are high because these functions take far too long. For example, it can take anywhere from about a month to almost half a year for an ITO to certify that a new server model is acceptable for use across a large corporation's data centers. (This happens *before* significant application-level testing!)

It would be useful to know why the process takes so long, but I have been unable to discover any careful categorization of the time spent. (This itself would be a good

research project.) In informal conversations, I learned that a major cause of the problem is the huge range of operating system versions that must be supported; although ITOs try to discourage the use of obsolete or modified operating systems, they must often support applications not yet certified to use the most up-to-date, vanilla release. The large number of operating system versions multiplies the amount of testing required.

Virtual machine technology can reduce the multiplication effect, since VMs impose regularity above the variability of hardware platforms. Once a set of operating systems has been tested on top of a given VM release, and that release has been tested on the desired hardware, the ITO can have some faith that any of these operating systems will probably work on that hardware with that VM layered in between. However, this still leaves open the problem of multiple versions of the VM software, and VMs are not always desirable (e.g., for performance reasons).

The long lead time for application deployment and upgrades contributes directly to business rigidity. A few companies (e.g., Amazon, Yahoo, Google) are considered "agile" because their IT systems are unusually flexible, but most large organizations cannot seem to solve this problem.

4 Where has OS research gone wrong?

At this point, the reader mutters "But, but, but ... we operating system researchers are all about 'flexibility!'." Unfortunately, it has often been the wrong kind of flexibility.

To oversimplify a bit, the two major research initiatives to provide operating system flexibility have been microkernels (mix & match services outside the kernel) and extensible operating systems (mix & match services inside the kernel). These initiatives focussed on increasing the flexibility of system-level services available to applications, and on flexibility of operating system implementation. They did not really focus on increasing application-level flexibility (perhaps because we have no good way to measure that; see Section 6).

Outside of a few niche markets, neither microkernels nor extensible operating systems have been successful in the enterprise IT market. The kinds of flexibility offered by either technology seems to create more problems than they solve:

- The ITO (or system vendor) ends up with no idea what daemons or extensions the user systems are actually running, which makes support *much* harder. It is hard to point the finger when something goes wrong.
- The ITO has no clear definition of what configurations have been tested, and ends up with a combinatorial explosion of testing problems. ("Safe" extensions are not really safe at the level of the whole IT system; they just avoid the obvious interface violations. Bad interactions through good interfaces are not checked.)
- The ITO has more difficulty maintaining a consistent execution environment for applications, which means that application deployment is even more difficult.

One might argue that increased flexibility for the operating system designer can too easily lead to *decreased* flexibility for the operating system user; it's easier to build novel applications on bedrock than on quicksand.

In contrast, VM research has led to market success. The term "virtual machine" is applied both to systems that create novel abstract execution environments (e.g., Java bytecodes) and those that expose a slightly abstract view of a real hardware environment (e.g., VMware or Xen [4]). The former model is widely seen as encouraging application portability through the provision of a standardized foundation; the latter model has primarily been viewed by researchers as supporting better resource allocation, availability, and manageability. But the latter model can also be used to standardize execution environments (as exemplified by PlanetLab [5] or Xenoservers [7]); VMs do aid overall IT flexibility.

5 How could OS research help?

In this section I suggest a few of the many operating system research problems that might directly or indirectly improve support for business change.

5.1 OS support for guaranteed sameness

If uncontrolled or unexpected variation in the operating environment is the problem, can we stamp it out? That is, without abolishing all future changes and configuration options, can we prevent OS-level flexibility from inhibiting business-level flexibility?

One way to phrase this problem is: can we *prove* that two operating environments are, in their aspects that affect application correctness, 100.00000000% identical? That is, in situations where we do not want change, can we formally prove that we have "sameness"?

Of course, I do not mean that operating systems or middleware should never be changed at all. Clearly, we want to allow changes that fix security holes or other bugs, improvements to performance scalability, and other useful changes that are irrelevant to the stability of the application. I will use the term "operationally identical" to imply a notion of useful sameness that is not too rigid.

If we could prove that host A is operationally identical to host B, then we could have more confidence that an application, once tested on host A, would run correctly on host B. More generally, A and B could each be clusters rather than individual hosts.

Similarly, if we could prove that A_0 is operationally

identical to $A_1, ..., A_n$, an application tested only on A_0 might be safe to deploy on $A_1, ..., A_n$.

It seems likely that this would have to be a formal proof, or else an ITO probably would not trust it (and would have to fall back on time-consuming traditional testing methods). However, formal proof technology typically has not been accessible to non-experts. Perhaps by restricting an automated proof system to a sufficiently narrow domain, it could be made accessible to typical IT staff.

On the other hand, if an automated proof system fails to prove that A and B are identical, that should reveal a specific aspect (albeit perhaps one of many) in which they differ. That could allow an ITO either to resolve this difference (e.g., by adding another configuration item to an installation checklist) or to declare it irrelevant for a specific set of applications. The proof could then be reattempted with an updated "stop list" of irrelevant features.

It is vital that a sameness-proof mechanism cover the entire operating environment, not just the kernel's API. (Techniques for sameness-by-construction might be an alternative to formal proof of sameness, but it is hard to see how this could be applied to entire environments rather than individual operating systems.) Environmental features can often affect application behavior (e.g., the presence and configuration of LDAP services, authentication services, firewalls, etc. [24]). However, this raises the question of how to define "the entire environment" without including irrelevant details, such as specific host IP addresses, and yet without excluding the relevant ones, such as the correct CIDR configuration.

The traditional IT practice of insisting that only a few configuration variants are allowed can ameliorate the sameness problem at time of initial application deployment. However, environments cannot remain static; frequent mandatory patches are the norm. But it is hard to ensure that every host has been properly patched, especially since patching often affects availability and so must often be done in phases. For this and similar reasons, sameness can deteriorate over time, which suggests that a sameness-proof mechanism would have to be reinvoked at certain points.

Business customers are increasingly demanding that system vendors pre-configure complex systems, including software installation, before shipping them. This can help establish a baseline for sameness, but vendor processes sometimes change during a product lifetime. A sameness-proof mechanism could ensure that vendor process changes do not lead to environmental differences that would affect application correctness.

5.2 Quantifying the value of IT

A business cannot effectively manage an IT system when it does not know how much business value that system generates. Most businesses can only estimate this value, for lack of any formal way to measure it. Similarly, a business that cannot quantify the value of its IT systems might not know when it is in need of IT-level change.

ITOs typically have budgets separate from the profit-and-loss accountability of customer-facing divisions, and thus have much clearer measures of their costs than of their benefits to the entire business. An ITO is usually driven by its local metrics (cost, availability, number of help-desk calls handled per hour). ITOs have a much harder time measuring what value its users gain from specific practices and investments, and what costs are absorbed by its users. As a result, large organizations tend to lack global rationality with respect to their IT investments. This can lead to either excessive or inadequate caution in initiating business changes. (It is also a serious problem for accountants and investors, because "the inability to account for IT value means [that it is] not reflected on the firm's [financial reports]", often creating significant distortions in these reports [23].)

Clearly, most business value is created by applications, rather than by infrastructure and utilities such as backup services [23]. This suggests that most work on value-quantification must be application-specific; why should we think operating system research has anything to offer?

One key issue is that accounting for value, and especially in ascribing that value to specific IT investments, can be quite difficult in the kinds of heavily shared and multiplexed infrastructures that we have been so successful at creating. Technologies such as timesharing, replication, DHTs, packet-switched networks and virtualized CPUs, memory, and storage make value-ascription hard.

This suggests that the operating environment could track application-level "service units" (e.g., requests for entire Web pages) along with statistics for response time and resource usage. Measurements for each category of service unit (e.g., "catalog search" or "shopping cart update") could then be reported, along with direct measurements of QoS-related statistics and of what IT assets were employed. The Resource Containers abstract [2] provides a similar feature, but would have to be augmented to include tracking information and to span distributed environments. Magpie [3] also takes some steps in this direction.

Accounting for value in multiplexed environments is not an easy problem, and it might be impossible to get accurate answers. We might be limited to quantifying only certain aspects of IT value, or we might have to settle for measuring "negative value," such as the opportunity cost of unavailability or delay. (An IT change that reduces a delay that imposes a clear opportunity cost has a fairly obvious value.)

5.3 Pricing for software licenses

Another value-related problem facing ITOs is the cost of software licenses. License fees for many major software products are based on the number of CPUs used, or on total CPU capacity. It is now widely understood that this simple model can discourage the use of technologies that researchers consider "obviously" good, including multi-core and multi-threaded CPUs, virtualized hardware, grid computing [22], and capacity-on-demand infrastructure. Until software vendors have a satisfactory alternative, this "tax on technology innovation with little return" [8] could distort ITO behavior, and inhibit a "business change" directly relevant to our field (albeit a one-time change).

The solution to the software pricing crisis (assuming that Open Source software cannot immediately fill all the gaps) is to price based on value to the business that buys the software; this provides the right incentives for both buyer and seller. (Software vendors might impose a minimum price to protect themselves against incompetent customers.)

Lots of software is already priced per-seat (e.g., Microsoft Office and many CAD tools) or per-employee (e.g., Sun's Java Enterprise System [18]), but these models do not directly relate business value to software costs, and might not extend to software for service-oriented computing.

Suppose one could instead track the number of application-level service units successfully delivered to users within proscribed delay limits; then application fees could be charged based on these service units rather than on crude proxies such as CPU capacity. Also, software vendors would have a direct incentive to improve the efficiency of their software, since that could increase the number of billable service units. Such a model would require negotiation over the price per billable service unit, but by negotiating at this level, the software buyer would have a much clearer basis for negotiation.

Presumably, basing software fees on service units would require a secure and/or auditable mechanism for reporting service units back to the software vendor. This seems likely to require infrastructural support (or else buyers might be able to conceal service units from software vendors). See Section 5.5 for more discussion of auditability.

One might also want a system of trusted third-party brokers to handle the accounting, to prevent software vendors from learning too much, too soon, about the business statistics of specific customers. A broker could anonymize the per-customer accounting, and perhaps randomly time-shift it, to provide privacy about business-level details while maintaining honest charging.

5.4 Name spaces that don't hinder organizational change

Operating systems and operating environments include lots of name spaces; naming is key to much of computer systems design and innovation.[1] We name system objects (files, directories, volumes, storage servers, storage services), network entities (links, switches, interfaces, hosts, autonomous systems), and abstract principals (users, groups, mailboxes, messaging servers).

What happens to these name spaces when an organizations combine or establish a new peering relationship? Often these business events lead to name space problems, either outright conflicts (e.g., two servers with the same hostname) or more abstract conflicts (e.g., different designs for name space hierarchies). Fixing these conflicts is painful, slow, error-prone, and expensive. Alan Karp has articulated the need to "design for consistency under merge" to avoid these conflicts [10].

And what happens to name spaces when an organization is split (e.g., as in a divestiture)? Some names might have to be localized to one partition or another, while other names might have to continue to resolve in all partitions. One might imagine designing a naming system that supports "completeness after division," perhaps through a means to tag certain names and subspaces as "clonable."

When systems researchers design new name spaces, we cannot focus only on traditional metrics (speed, scale, resiliency, security, etc.); we must also consider how the design supports changes in name-space scope.

5.5 Auditability for outsourcing

IT practice increasingly tends towards outsourcing (distinct from "offshoring") of critical business functions. Outsourcing can increase business flexibility, by giving a business immediate access to expertise and sometimes by better multiplexing of resources, but it requires the business to *trust* the outsourcing provider. Outsourcing exposes the distinction between *security* and *trust*. Security is a technical problem with well-defined specifications, on which one can, in theory, do mathematical proofs. Trust is a social problem with shifting, vaque requirements; it depends significantly on memory of past experiences. Just because you can prove to yourself that your systems are secure and reliable does not mean that you can get your customers to entrust their data and critical operations to you.

This is a variant of what economists call the "principal-agent problem." In other settings, a principal could establish its trust in an agent using a third-party auditor, who has sufficient access to the agent's environment to check for evidence of incorrect or improper practices. The auditor has expertise in this checking process that the principal does not, and also can investigate agents who serve multiple principals without fear of in-

formation leakage.

Pervasive outsourcing might therefore benefit from infrastructural support for auditing; i.e., the operating environment would support monitoring points to provide "sufficient access" to third-party auditors. Given that much outsourcing will be done at the level of operating system interfaces, some of the auditing support will come from the operating system. For example, the system might need to provide evidence to prove that principal A cannot possibly see the files of principal B, and also that this has never happened in the past.

6 Operating outside our comfort zone

The problems of enterprise computing, and especially of improving business-level (rather than IT) metrics, is far outside the comfort zone of most operating system researchers. Problems include

- The applications are not the ones we use or write ourselves; it is hard to do operating system research using applications one does not understand.
- Most of these applications are not Open Source; researchers cannot afford them, and some vendors ban unauthorized benchmarking.
- The applications can be hard to install. A typical SAP installation might involve millions of dollars of consultant fees over months or even years to customize it [11].
- We do not have a good description of "real workloads" for these applications.

In addition, many of the problems inhibiting business change are cultural, not technical. That does not mean that we are excused from addressing the technical challenges, but this is an engineering science, so our results need to respect the culture in which they would be used. That means that computer science researchers need to learn about that culture, not just complain about it.

6.1 What about metrics?

Perhaps the biggest problem is that we lack quantified metrics for things like "business flexibility." (Low-level flexibility metrics, such as "time to add a new device driver to the kernel," are not the right concept.) Lacking the metrics, we cannot create benchmarks or evaluate our ideas.

Rob Pike has argued that "In a misguided attempt to seem scientific, there's too much measurement: performance minutiae and bad charts. ... Systems research cannot be just science; there must be engineering, design, and art." [20]. But we *must* measure, because otherwise we cannot establish the value of IT systems and processes; however, we should not measure the wrong things ("performance minutiae") simply because those are the easiest for us to measure.

Metrics for evaluating how well IT systems support business change will not be as simple as, for example, measuring Web server transaction rates, for at least two reasons. First, because such evaluations cannot be separated from context; successful change inevitably depends on people and their culture, as well as on IT. Second, because business change events, while frequent enough to be problematic, are much rarer and less repeatable than almost anything else computer scientists measure. We will have to learn from other fields, such as human factors research and economics, ways to evaluate how IT systems interact with large organizations.

I will speculate on a few possible metrics:

- **For software deployment**: It might be tempting to simply measure the time it takes to deploy an application once it has been tested. However, such timing often depends too much on uncontrollable variables, such as competing demands on staff time. A more repeatable metric would be the number of new problems found in the process of moving a "working" application from a test environment to a production environment. The use of bug rates as a metric was proposed in a similar context by Doug Clark [6], who pointed out that what matters is not reducing the total number of bug reports, but finding them as soon as possible, and before a product ships to customers.

  Nagaraja *et al.* reported on small-scale measurements of how frequently operators made mistakes in reconfiguring Internet applications [16]. They described a technique to detect many such errors automatically, using parallel execution of the old system and the new system, comparing the results, with the new system isolated to prevent any errors from becoming visible. Their approach might be generalizable to testing for environmental sameness.

  One might also crudely measure a system's support for deployment of updated applications by subjecting an application to increasingly drastic changes until something breaks. For example, perhaps the operating environment can support arbitrary increases in the number of server instances for an application, but not in the number of geographically separated sites.
- **For quantifying IT value**: Suppose that an enterprise's IT systems generated estimates of their value. One way to test these estimates would be to compare their sum to the enterprise's reported revenue, but this probably would not work: revenue reports are too infrequent and too arbitrary, and it would require nearly complete value-estimation coverage over all IT systems. Instead, one might be able to find correlations between the IT-value estimates from distinct systems and the short-term per-product revenue metrics maintained by many businesses. If the correlations can be used for prediction (e.g., they persist after a system improvement) then they would validate

the IT-value estimates.

In the end, many important aspects of IT flexibility will never be reduced to simple, repeatable metrics. We should not let this become an excuse to give up entirely on the problem of honest measurement.

7 Grand Challenge ... or hopeless cause?

Section 6 describes some daunting problems. How can we possibly do research in this space? I think the answer is "because we must." Support for CS research, both from government and industry, is declining [9, 19]. If operating system research cannot help solve critical business problems, our field will shrink.

The situation is not dire. Many researchers are indeed addressing business-level problems. (Space prohibits a lengthy description of such work, and it would be unfair to pick out just a few.) But I think we must do better at defining the problems to solve, and at recognizing the value of their solution.

Acknowledgments

In preparing this paper, I had help from many colleagues at HP, including Martin Arlitt, Mary Baker, Terence Kelly, Bill Martorano, Jerry Rolia, and Mehul Shah. The HotOS reviewers also provided useful feedback.

References

[1] Q&A with Robert Napier, CIO at Hewlett-Packard. *CIO Magazine*, Sep. 15 2002. http://www.cio.com/archive/091502/napier.html.

[2] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. OSDI*, pages 45–58, New Orleans, LA, Feb. 1999.

[3] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modeling. In *Proc. 6th OSDI*, pages 259–272, San Francisco, CA, Dec. 2004.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. SOSP-19*, pages 164–177, 2003.

[5] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating System Support for Planetary-Scale Services. In *Proc. NSDI*, pages 253–266, San Francisco, CA, Mar. 2004.

[6] D. W. Clark. Bugs are Good: A Problem-Oriented Approach to the Management of Design Engineering. *Research-Technology Management*, 33(3):23–27, May-June 1990.

[7] K. Fraser, S. M. Hand, T. L. Harris, I. M. Leslie, and I. A. Pratt. The Xenoserver computing infrastructure. Tech. Rep. UCAM-CL-TR-552, Univ. of Cambridge, Computer Lab., Jan. 2003.

[8] Garner, Inc. Gartner says cost of software licenses could increase by at least 50 percent by 2006. http://www3.gartner.com/press_releases/asset_115090_11.html, Nov. 2004.

[9] P. Harsha. NSF budget takes hit in final appropriations bill. *Computing Research News*, 17(1), Jan. 2005. http://www.cra.org/CRN/articles/jan05/harsha.html.

[10] A. H. Karp. Lessons from E-speak. Tech. Rep. HPL-2004-150, HP Labs, Sep. 2004. http://www.hpl.hp.com/techreports/2004/HPL-2004-150.html.

[11] C. Koch. Lump it and like it. *CIO Magazine*, Apr. 15 1997. http://www.cio.com/archive/041597/lump.html.

[12] C. Koch. AT&T Wireless Self-Destructs. *CIO Magazine*, Apr. 15 2004. http://www.cio.com/archive/041504/wireless.html.

[13] C. Koch. When bad things happen to good projects. *CIO Magazine*, Dec. 1 2004. http://www.cio.com/archive/120104/contingencyplan.html.

[14] J. C. McGroddy and H. S. L. Editors. A Review of the FBI's Trilogy Information Technology Modernization Program. http://www7.nationalacademies.org/cstb/pub_fbi.html, 2004.

[15] S. Minton, E. Opitz, J. Orozco, F. Chang, S. J. Frantzen, G. Koch, M. Coughlin, T. G. Copeland, and A. Tocheva. Worldwide IT Spending 2004-2008 Forecast. IDC document #32321, Dec. 2004.

[16] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proc. OSDI*, pages 61–76, San Francisco, CA, Dec 2004.

[17] National Commission on Terrorist Attacks Upon the United States. Final report, 2004.

[18] J. Niccolai and D. Tennant. Sun pricing model impresses users. *ComputerWeekly.com*, Sep. 2003. http://www.computerweekly.com/Article125119.htm.

[19] M. Pazzani, K. Abdali, G. Andrews, and S. Kim. Cise update: Adjusting to the increase in proposals. *Computing Research News*, 16(5), Nov. 2004. http://www.cra.org/CRN/articles/nov04/pazzani.html.

[20] R. Pike. Systems software research is irrelevant. http://herpolhode.com/rob/utah2000.pdf, 2000.

[21] D. Rohrer. Personal communication, 2005.

[22] P. Thibodeau. Software licensing emerges as grid obstacle. *ComputerWorld*, May 2004. http://www.computerworld.com/printthis/2004/0,4814,93526,00.html.

[23] J. Tillquist and W. Rodgers. Using asset specificity and asset scope to measure the value of IT. *Comm. ACM*, 48(1):75–80, Jan. 2005.

[24] J. Wilkes, J. Mogul, and J. Suermondt. Utilification. In *Proc. 11th SIGOPS European Workshop*, Leuven, Belgium, Sep. 2004.

Notes

[1] I think Roger Needham said that (more eloquently), but I haven't been able to track down a quote.

Designing Controllable Computer Systems

Christos Karamanolis
Hewlett-Packard Labs

Magnus Karlsson
Hewlett-Packard Labs

Xiaoyun Zhu
Hewlett-Packard Labs

Abstract

Adaptive control theory is emerging as a viable approach for the design of self-managed computer systems. This paper argues that the systems community should not be concerned with the design of adaptive controllers—there are off-the-shelf controllers that can be used to tune any system that abides by certain properties. Systems research should instead be focusing on the open problem of designing and configuring systems that are amenable to dynamic, feedback-based control. Currently, there is no systematic approach for doing this. To that aim, this paper introduces a set of properties derived from control theory that controllable computer systems should satisfy. We discuss the intuition behind these properties and the challenges to be addressed by system designers trying to enforce them. For the discussion, we use two examples of management problems: 1) a dynamically controlled scheduler that enforces performance goals in a 3-tier system; 2) a system where we control the number of blades assigned to a workload to meet performance goals within power budgets.

1 Introduction

As the size and complexity of computer systems grow, system administration has become the predominant factor of ownership cost [6] and a main cause for reduced system dependability [14]. The research community has recognized the problem and there have been several calls to action [11, 17]. All these approaches propose some form of self-managed, self-tuned systems that aim at minimizing manual administrative tasks.

As a result, computers are increasingly designed as *closed-loop systems*: as shown in Figure 1, a controller automatically adjusts certain parameters of the system, on the basis of feedback from the system. Examples of such closed-loop systems aim at managing the energy consumption of servers [4], automatically maximizing the utilization of data centers [16, 18], or meeting performance goals in file servers [9], Internet services [10, 12] and databases [13].

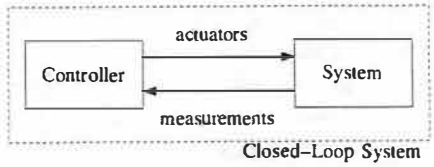


Figure 1: A closed-loop system.

When applying dynamic control, it is important that the resulting closed-loop system is stable (does not exhibit large oscillations) and converges fast to the desired end state. Many existing closed-loop systems use ad-hoc controllers and are evaluated using experimental methods. We claim that a more rigorous approach is needed for designing dynamically controlled systems. In particular, we advocate the use of *control theory*, because it results in systems that can be shown to work beyond the narrow range of a particular experimental evaluation. Computer system designers can take advantage of decades of experience in the field and can apply well-understood and often automated methodologies for controller design.

However, we believe that systems designers should not be concerned with the design of controllers. Control theory is an active research field on its own, which has produced streamlined control methods [2] or even off-the-shelf controller implementations [1] that systems designers can use. Indeed, we show that many computer management problems can be formulated so that standard controllers can be applied to solve them. Thus, the systems community should stick with systems design; in this case, systems that are amenable to dynamic, feedback-based control. That is, provide the necessary tunable system parameters (*actuators*) and export the appropriate feedback metrics (*measurements*), so that an off-the-shelf controller can be applied without destabilizing the system, while it ensures fast convergence to the desired goals. Traditionally, control theory has been concerned with systems that are governed by laws of physics (e.g., mechanical devices), thus allowing to make assertions about the existence or not of certain

properties. This is not necessarily the case with software systems. We have seen in practice that checking whether a system is controllable or, even more, building controllable systems is a challenging task often involving non-intuitive analysis and system modifications.

As a first step in addressing the latter problem, this paper proposes a set of *necessary and sufficient properties* that any system must abide by to be controllable by a standard adaptive controller that needs little or no tuning for the specific system. These properties are derived from the theoretical foundations of a well-known family of adaptive controllers. The paper discusses the intuition and importance of these properties from a systems perspective and provides insights about the challenges facing the designer that tries to enforce them. The discussion has been motivated by lessons learned while designing self-managed systems for an adaptive enterprise environment [17]. In particular, we elaborate on the discussion of the properties with two very diverse management problems: 1) enforcing soft performance goals in networked service by dynamically adjusting the shares of competing workloads; 2) controlling the number of blades assigned to a workload to meet performance goals within power budgets.

2 Dynamic Control

Many computer management problems can be cast as on-line optimization problems. Informally speaking, the objective is to have a number of measurements obtained from the system converge to the desired goals by dynamically setting a number of system parameters (actuators). The problem is formalized as an *objective function* that has to be minimized. A formal problem specification is outside the scope of this paper. The key point here is that there are well-understood, standard controllers that can be used to solve such optimization problems. Existing research has shown that, in the general case, *adaptive* controllers are needed to trace the varying behavior of computer systems and their changing workloads [9, 12, 18].

2.1 Self-tuning regulators

For the discussion in this paper, we focus on one of the best-known families of adaptive controllers, namely *Self-Tuning Regulators* (STR)) [2], that have been widely used in practice to solve on-line optimization control problems. The term "self-tuning" comes from the fact that the controller parameters are automatically tuned to obtain the desired properties of the closed-loop system. The design of closed-loop systems involves many tasks such as modeling, design of control law, implementation, and validation. STR controllers aim at automating these tasks. Thus, they can be used out-of-the-box for many practical cases. Other types of adaptive controllers proposed in the literature usually require more intervention by the designer.

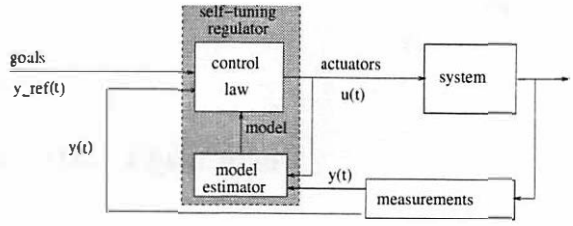


Figure 2: The components of a self-tuning regulator.

As shown in Figure 2, an STR consists of two basic modules: the *model estimator* module on-line estimates a model that describes the current measurements from the system as a function of a finite history of past actuator values and measurements; that model is then used by the *control law* module that sets the actuator values. We propose using a linear model of the following form for model estimation in the STR:

$$y(t) = \sum_{i=1}^{n} A_i y(t-i) + \sum_{i=0}^{n-1} B_i u(t-i-d_0) \qquad (1)$$

where $y(t)$ is a vector of the N measurements at time t and $u(t)$ is a vector capturing the M actuator settings at time t. A_i and B_i are the model parameters, with dimensions compatible with those of vectors $y(t)$ and $u(t)$. n is the *model order* that captures how much history the model takes into account. Parameter d_0 is the delay between an actuation and the time the first effects of that actuation are observed on the measurements. The unknown model parameters A_i and B_i are estimated using *Recursive Least-Squares* (RLS) estimation [7]. This is a standard, computationally fast estimation technique that fits (1) to a number of measurements, so that the sum of squared errors between the measurements and the model predictions is minimized. For the discussion in this paper, we focus on discrete-time models. One time unit in this discrete-time model corresponds to an invocation of the controller, i.e., sampling of system measurements, estimation of a model, and setting the actuators.

Clearly, the relation between actuation and observed system behavior is not always linear. For example, while throughput is indeed a linear function of the share of resources (e.g., CPU cycles) assigned to a workload, the relation between latency and resource share is nonlinear as Little's law indicates. However, even in the case of nonlinear metrics, a linear model is often a good-enough local approximation to be used by a controller [2], as the latter usually only makes small changes to actuator settings. The advantage of using linear models is that they can be estimated in computationally efficient ways. Thus, they result in tractable control laws and they admit simpler analysis including stability proofs for the closed-loop system.

The control law is essentially a function that, based on the estimated system model (1) at time t, decides what the actuator values $u(t)$ should be to minimize the objective function. The STR derives $u(t)$ from a closed-form expression as a function of previous actuations, previous measurements, model parameters and the reference values (goal) for the measurements. Details of the theory can be found in Åström *et al.* [2]. From a systems perspective, the important point is that these are computationally efficient calculations that can be performed on-line. Moreover, an STR requires little system-specific tuning as it uses a dynamically estimated model of the system and the control law automatically adapts to system and workload dynamics.

2.2 Properties of controllable systems

For the aforementioned process to apply and for the resulting closed-loop system to be stable and to have predictable convergence time, control theory has derived a list of *necessary and sufficient* properties that the target system must abide by [2, 7]. In the following paragraphs, we interpret these theoretical requirements into a set of system-centric properties. We provide guidelines about how one can verify whether a property is satisfied and what are the challenges for enforcing them.

C.1. Monotonicity. *The elements of matrix B_0 in (1) must each have a known sign that remains the same over time.*

The intuition behind this property is that the real (non estimated) relation between any actuator and any measurement must be monotonic and of known sign. This property usually refers to some physical law. Thus, it is generally easy to check for it and get the signs of B_0. For example, in the long term, a process with a large share of CPU cycles gets higher throughput and lower latency than one with a smaller share.

C.2. Accurate models. *The estimated model (1) is a good-enough local approximation of the system's behavior.*

As discussed, the model estimation is performed periodically. A fundamental requirement is that the dynamic relation between actuators and measurements is captured sufficiently by the model around the current operating point of the system. In practice, this means that the estimated model must track only real system dynamics. We use the term *noise* to describe deviations in the system behavior that are not captured by the model. It has been shown that to ensure stability in *linear* systems where there is a known upper bound on the noise amplitude, the model should be updated only when the model error is twice the noise bound [5]. The theory is more complicated for nonlinear systems [15], but the above principle can be used as a rule of thumb in that case too. There are a number of sources for the aforementioned noise:

1. System dynamics that have a frequency higher than that of sampling in the system, especially when one measures instantaneous values instead of averages over the sampling interval.

2. Sudden transient deviations from the operating range of the system. For example, rapid latency fluctuations because of contention on a shared network link.

3. A fundamentally volatile relation between certain actuators and measurements. One example is the relation between resource shares and provided throughput. When the aggregate throughput of the system oscillates a lot (as is often the case in practice), this relation is volatile. Instead, a more stable relation could be expressed as the fraction of the total system throughput received by a workload as a function of share.

4. Quantization errors when a linear model is used to approximate locally in an operating range the behavior of a nonlinear system.

In fact, a tiny actuation error has often to be introduced, so that the system is excited sufficiently for a good model to be derived. In other words, the system is forced to slightly deviate from its operating point to derive a linear model approximation (you need two points to draw a line). It is the controller that typically introduces such small perturbations for modeling purposes.

Picking actuators and measurement metrics that result in stable, ideally linear, relations is one of the most challenging and important tasks in the design of a controllable system, as we discuss in Section 3. The following two properties are also related to the requirement for accurate models.

C.3. Known system delay. *There is a known upper bound d_0 on the time delay in the system.*

C.4. Known system order. *There is a known upper bound n on the order of the system.*

Property C.3 ensures that the controller knows when to expect the first effects of its actuations, while C.4 ensures that the model remembers sufficiently many prior measurements ($y(t)$) and actuations ($u(t)$) to capture the dynamics of the system. These properties are needed for the controller to be able to observe the effects of its actuations and then attempt to correct any error in subsequent actuations. If the model order was less than the actual system order, then the controller would not be aware of some of the causal relations between actuation and measurements in the system. The values of d_0 and n are derived experimentally. The designer is faced with a trade-off: On one hand, the values of d_0 and n must be sufficiently high to capture as much as possible of the causal relations between actuation and measurements. On the other hand, a high d_0 implies a slow-responding controller and a high n increases

are ideal values.

C.5. Minimum phase. *Recent actuations have higher impact on the measurements than older actuations do.*

A minimum phase system is basically one for which the effects of an actuation can be corrected or canceled by another, later actuation. It is possible to design STRs that deal with non-minimum phase systems, but they involve experimentation and non-standard design processes. In other words, without the minimum phase requirement, we cannot use off-the-shelf controllers. Typically, physical systems are minimum phase—the causal effects of events in the system fade as time passes by. Sometimes, however, this is not the case with computer systems, as we see in Section 3. To ensure this property, a designer must re-set any internal state that reflects older actuations. Alternatively, the sample interval can always be increased until the system becomes minimum phase. Consider, for example, a system where the effect of an actuation peaks after three sample periods. By increasing the sample period threefold, the peak is now contained in the first sample period, thus abiding by this property. Increasing the sample interval should only be a last resort, as longer sampling intervals result in slower control response.

C.6. Linear independence. *The elements of each of the vectors $y(t)$ and $u(t)$ must be linearly independent.*

Unless this property holds, the quality of the estimated model is poor: the predicted value for $y(k)$ may deviate considerably from the actual measurements. The reason for this requirement is that some of the calculations in RLS involve matrix inversion. When C.6 is not satisfied, there exists a matrix internal to RLS that is singular or close to singular. When inverted, that matrix contains very large numbers, which in combination with the limited resolution of floating point arithmetic of a CPU, result in models that are wrong. Note that the property does not require that the elements in $u(k)$ and in $y(k)$ are completely non-correlated; they must not be linearly correlated. Often, simple intuition about a system may be sufficient to ascertain if there are linear dependencies among actuators, as we see in Section 3.

C.7. Zero-mean measurements and actuator values. *The elements of each of the vectors $y(k)$ and $u(k)$ should have a mean value close to 0.*

If the actuators or the measurements have a large constant component to them, RLS tries to accurately predict this constant component and may thus miss to capture the comparably small changes due to actuation. For example, when the measured latency (in ms) in a system varies in the $[1000, 1100]$ range depending on the share of resources assigned to a workload, the model estimator would not accurately capture the relatively small changes due to the share actuation. If there is a large constant component in the measurements and it is known, then it can be simply deducted. If the constant component is unknown, then it can be easily estimated using a moving average. Problems may arise if this constant value changes rapidly, for example when a workload rapidly alternates between being disk-bound and cache-bound resulting in more than an order of magnitude difference in measured latency. In that case, it is probably better to search for a new actuator and measurement combination.

C.8. Comparable magnitudes of measurements and actuator values. *The values of the elements in $y(k)$ and $u(k)$ should not differ by more than one order of magnitude.*

If the measurement values or the actuator values differ considerably, then RLS results in a model that captures more accurately the elements with the higher values. There are no theoretical results to indicate the threshold at which RLS starts producing bad models. Instead, we have empirically found that the quality of models estimated by RLS in a control loop start degenerating fast after a threshold of one order of magnitude difference. This problem can be solved easily by scaling the measurements and actuators, so that their values are comparable. This scaling factor can also be estimated using a moving average.

3 Case studies

In this section, we illustrate the systems aspects of the previous properties and the wide applicability of the approach, with two examples of management problems.

3.1 Controllable Scheduler

Here, we consider a 3-tier e-commerce service that consists of a web server, an application server and a database. A scheduler is placed on the network path between the clients and the front end of the service. It intercepts client http requests and re-orders or delays them to achieve differentiated quality of service among the clients. The premise is that the performance of a client workload varies in a predictable way with the amount of resources available to execute it. The scheduler enforces approximate proportional sharing of the service's capacity to serve requests (throughput) aiming at meeting the performance goals of the different client workloads. In particular, we use a a variation of *Weighted Fair Queuing* (WFQ) that works in systems with high degree of concurrency.

However, given the dynamic nature the system and the workloads, the same share of the service's capacity does not always result in the same performance; e.g., a 10% share for some client may result in a average latency of 100 ms at one point in time and in 250 ms a few seconds later. Thus, shares have to be adjusted dynamically to enforce the workload performance objectives. The on-line optimization problem that needs to be solved here is to set

the shares of the different competing workloads so that the difference between actual measurements and performance goals is minimized overall workloads (possibly considering priorities among workloads).

According to the terminology of the previous section, the 3-tier service including the scheduler is our system; the workload shares are the actuators $u(t)$ and the performance measurements (latency or throughput) of the workloads are $y(t)$; the performance goals for the workloads are captured by $y_{ref}(t)$. However, when used in tandem with the controller, the scheduler could not be tuned to meet the workload performance goals in the service operating under a typical workload. The closed-loop system often became unstable and would not converge to the performance goals.

While investigating the reasons of this behavior, we observed that actuation (setting workload shares) by the controller would often have no effect in the system. As a result, the controller would try more aggressive actuation which often led to oscillations. Going through the properties of Section 2, we realized that C.5 (minimum phase) was violated. WFQ schedulers dispatch requests for processing in ascending order of *tags* assigned to the requests upon arrival at the scheduler; the tags reflect the relative share of each workload. However, when the shares vary dynamically, the tags of queued requests are not affected. Thus, depending on the number of queued requests in the scheduler, it may take arbitrarily long for the new shares to be reflected on dispatching rates. In other words, there is no way to compensate for previous actuations. For the same reasons, properties C.3 and C.4 (known bounds on delay and order) are not satisfied either. One way to address this problem is by increasing the sampling period. However, this would not work in general because the number of queued requests with old tags depends on actual workload characteristics and is not necessarily bounded. So, instead, we looked into modifying the system. In particular, we modified the basic WFQ algorithm to recalculate the tags of queued requests every time shares change. Thus, controller actuations are reflected immediately on request dispatching. After this modification, properties C.3 – C.5 are satisfied and $d_0 = 1$, $n = 1$ for a sampling period of 1 second.

Another, minor problem with the scheduler was due to the inherent linear dependency of any single share (actuator) to the other $N - 1$ shares: its value is 100% minus the sum of the others. As a result, property C.6 was violated. We addressed this problem by simply keeping only $N - 1$ actuators. The scheduler derived the value of the Nth actuator from that of the others.

The system abides by all other properties. Monotonicity (C.1) may not hold for a few sampling intervals, but it does hold on average in the long term. Moreover, we have seen that, with an estimation period of around 1 second, an online RLS estimator is able to trace the system dynamics

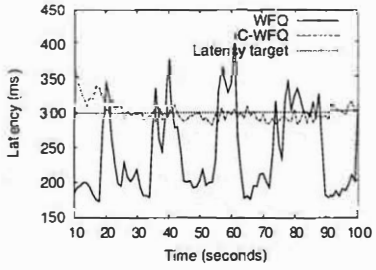


Figure 3: Using an STR to control workload shares in WFQ and C-WFQ. The graph depicts one of the workloads in the system. WFQ results in an unstable system that misses workload goals.

with locally linear models (C.2). The noise level in the measurements for the 3-tier service is at most 2% and thus we chose a model update threshold of 4%. Property C.7 (zero means) is easily satisfied by using a moving average to calculate on-line a constant factor which is then subtracted from the measurement values. Similarly, we use a moving average to estimate a normalization ratio for the measurements (C.8, value magnitudes).

Figure 3 illustrates the performance of the system with the conventional (WFQ) and the modified (C-WFQ) schedulers. The site hosted on the system is a version of the Java PetStore [8]. The workload applied to it mimics real-world user behavior, e.g., browsing, searching and purchasing, including the corresponding time scales and probabilities these occur with. The fact that WFQ is not controllable results in oscillations in the system and substantial deviations from the performance goals.

3.2 Trading off power and performance

In this case, the objective is to trade off power consumption and performance targets (both captured in $y(t)$) in a data center by controlling the number of blades dedicated to a workload (captured in $u(k)$). The on-line optimization aims at reducing the overall difference between $y(t)$ and the goals for performance and power consumption. In the case of power, the goal is zero consumption, i.e., minimization of the absolute value. All the data used for the discussion here are taken from Bianchini *et al.* [3].

Clearly, increasing the number of blades monotonically increases consumed power and delivered performance (C.1). When a new blade is added, there is a spike before power consumption settles to a new (higher) level. This suggests that it would be hard to satisfy C.2 (accurate models). However, other than this transient spike, the relation between power and the number of blades, and between performance and the number of blades is steady with an error of less than 5%. In order to abide by C.2, we can get rid off the initial spike in one of two ways: 1) by ignoring those power measurements, using a higher sample period,

e.g., of several seconds; 2) by automatically factoring in this spike in the model estimation by using a higher model order (n in C.4) with a sample period of just a few seconds. The value of d_0 (C.3) depends on the sampling period and on whether new blades have to be booted (higher d_0) or are stand-by (lower d_0). C.5 (minimum phase) is satisfied, as the effects of new settings (number of blades) override previous ones. C.7 and C.8 can also be trivially satisfied by using a moving average, as described in Section 2. Things are a little more subtle with C.6 (linear independence). In certain operating ranges, power and performance depend linearly on each other. In those cases, the controller should consider only one of these measurements as $y(k)$ to satisfy C.6.

4 Conclusion

Designing closed-loop systems involves two key challenges. First, rigorous controller design is a hard problem that has been an active research area for decades. The resulting theory and methodology are not always approachable by the systems community. However, certain management problems in computer systems can be formulated so that designers may use automated approaches for controller design or even use off-the-shelf controllers. Such problems include meeting performance goals [10], maximizing the utility of services, and improving energy efficiency [4]. It is an open issue how other problems, such as security or dependability objectives, can be formalized as dynamic control problems.

Thus, for a range of management problems, controller design can be considered a solved problem for the systems community. We should instead be focusing on a second challenge that is closer to our skill set. That is, how to design systems that are amenable to dynamic control. This paper discusses a canonical set of properties, derived from control theory, that any system should abide by to be controllable by a standard adaptive controller. Checking for these properties is not always an intuitive process. Even worse, enforcing them requires domain-specific expertise, as we saw with the two examples in Section 3. Developing a systematic approach for building controllable systems is an open problem that deserves further attention.

References

[1] ABB Automation Products. *Avant Controller 410*, version 1.5/2 edition, 2001.

[2] K. J. Åström and B. Wittenmark. *Adaptive Control*. Electrical Engineering: Control Engineering. Addison-Wesley Publishing Company, 2 edition, 1995. ISBN 0-201-55866-1.

[3] R. Bianchini and R. Rajamony. Power and Energy Management for Server Systems. *IEEE Computer*, 37(11):68–74, November 2004.

[4] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing Energy and Server Resources in Hosting Centres. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 103–116, Banff, Canada, October 2001.

[5] B. Egardt. *Stability of Adaptive Controllers*, volume 20. Springer-Verlag, 1979. ISBN 0-38709-646-9.

[6] J. Gray. A conversation with Jim Gray. *ACM Queue*, 1(4), June 2003.

[7] M. Honig and D. Messerschmitt. *Adaptive Filters: Structures, Algorithms, and Applications*. Kluwer Academic Publishers, Hingham MA, 1984. ISBN 0-898-38163-0.

[8] *Java PetStore*. www.middleware-company.com.

[9] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance Isolation and Differentiation for Storage Systems. In *International Workshop on Quality of Service (IWQoS)*, pages 67–74, Montreal, Canada, June 2004.

[10] M. Karlsson, X. Zhu, and C. Karamanolis. An Adaptive Optimal Controller for Non-Intrusive Performance Differentiation in Computing Services. In *IEEE Conference on Control and Automation (ICCA)*, Budapest, Hungary, June 2005. To appear.

[11] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, January 2003.

[12] Y. Lu, T. Abdelzaher, C. Lu, and G. Tao. An adaptive control framework for QoS guarantees and its application to differentiated caching services. In *International Workshop on Quality of Service (IWQoS)*, pages 23–32, Miami Beach, FL, May 2002.

[13] S. Parekh, K. Rose, Y. Diao, V. Chang, J. Hellerstein, S. Lightstone, and M. Huras. Throttling Utilities in the IBM DB2 Universal Database Server. In *American Control Conference (ACC)*, pages 1986–1991, Boston, MA, June 2004.

[14] D. Patterson. A new focus for a new century: availability and maintainability >> performance. Keynote speech at USENIX FAST, January 2002.

[15] J.-J. Slotine and W. Li. *Applied Nonlinear Control*. Prentice Hall, 1991. ISBN 0-13-040890-5.

[16] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 239–254, Boston, MA, December 2002.

[17] J. Wilkes, J. Mogul, and J. Suermondt. Utilification. In *11th ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[18] S. S. Xue Liu, Xiaoyun Zhu and M. Arlitt. Adaptive entitlement control of resource containers on shared servers. In *9th IFIP/IEEE International Symposium on Integrated Network Management (IM)*, Nice, France, May 2005.

Three research challenges at the intersection of machine learning, statistical induction, and systems

Moises Goldszmidt, Ira Cohen
Hewlett-Packard Labs
Palo Alto, CA

Armando Fox, Steve Zhang
Computer Science Department
Stanford University

Abstract

Recent research activity [2, 12, 27, 10, 1] has shown encouraging results for performance debugging and failure diagnosis and detection in systems by using approaches based on automatically inducing models and deriving correlations from observed data. We believe that maximizing the potential of this line of research will require surmounting some fundamental challenges arising not from the modeling techniques themselves, but specifically from the *application* of those techniques to real-world systems. We specifically formulate three challenges. First, as new data is collected from a system, previously-induced models must be continuously assessed and validated, with the ultimate aim of achieving online adaption to system changes. Second, human operators must be able to effectively interact with the models, including interpreting model findings to generate explanations, enabling human feedback to improve the models, and identifying false positives and missed detections. Third, it should be possible to formally manipulate "signatures" of system state as represented by these models, allowing us to query the system's past to identify recurring problems and manually annotate them with additional information. We contend that the specifics of this problem domain not only raise these challenges, but also provide the knowledge base from which to derive well-engineered solutions to them. We suggest some possible strategies for addressing each challenge and show how they arise in the context of a real example.

1 Introduction

The complexity of today's deployed software systems is staggering, as is the rate of growth of that complexity. In terms of lines of code, in the last ten years Linux has grown by a factor of 30 and Cisco IOS by a factor of 10 while Apache has grown by a factor of five in the last five years. The result is that more than 90% of a typical corporate IT budget is devoted to administration and maintenance of existing systems [11] whose complexity surpasses human operators' ability to diagnose and respond to problems rapidly and correctly [17, 26].

Fortunately, promising initial results have been reported in using automatically-induced probabilistic and machine-learning-based models for problem localization [10], performance debugging [2, 1], capacity planning, system tuning [38], detecting non-failstop failures [27], and attributing performance problems to specific low-level metrics [12], among others. These efforts differ in the specific techniques, models, and assumptions (we list some representative examples later), but the general approach may be summarized as follows: Collect raw data from the running system; automatically induce a model over this data; use the model to make inferences. We believe this general direction is extremely encouraging because the automatic construction of models from data brings the promise of rapid adaptation to system changes or to unanticipated conditions. Despite the differences among approaches, we expect that there will arise fundamental challenges that *any* effort utilizing statistical methods will have to confront. Given the successes so far, we detail in this paper three such challenges in hopes of guiding this line of research towards realizing its full potential.

Our challenges may be summarized as: Can we design effective procedures and algorithms that continuously and automatically test the validity of models against a dynamic environment? How can model findings be interpreted by the human operators of the system, e.g. identifying false positives, converting model findings to actionable information, and possibly accepting feedback from experts? How can we maintain a long term, indexable, and searchable history of system issues, annotated in some cases with diagnosis/repair action, to leverage past diagnosis efforts and enable use of similarity-based search techniques in order to identify recurring problems and group similar problem incidents into common "syndromes"?

To understand how these challenges arose, it is useful to review some concrete approaches, including their assumptions and methods (Section 2); we then explore each challenge in detail (Sections 3–5) and give an example of how the challenges is addressed in the context of a specific approach. We make concluding remarks in Section 6.

2 Early explorations

We highlight some recent specific successes of recent approaches that automatically induce models and correlate data from a running networked system. These approaches concentrate on transforming data into information that can be used to make decisions. Our intent is not to present a complete survey, but to outline the ways in which the different approaches map to real systems problems, the assumptions underlying this mapping, the consequences of violating those assumptions, and the similarities among the approaches that will motivate our fundamental challenges.

One set of approaches relies on modeling *normal* behavior and then identifying sufficiently large *deviations* as a possible indication of undesirable behavior. For example, the technique in [27] identifies rarely-occurring paths at runtime by using probabilistic grammars to model the distribution of *normal* paths of program execution at the software-module level. The assumption is that a sufficiently *anomalous* path may indicate a possible failure. When this assumption is violated, e.g. because a rare but legitimate code path is traversed, an automatic repair system might mistakenly take a repair action. The work discusses options for low-cost repair techniques that cause no harm if invoked by mistake, to mitigate such inevitable "false positives." In controlled experiments with realistic workloads, this technique detected 15% more failures than existing generic techniques; localization of these faults exhibits a classic recall/precision tradeoff, with false positive rates (1 − precision) approaching 20% for high values of recall, emphasizing the importance of dealing with false positives.

A second approach [12, 41] explicitly defines abnormal vs. normal behavior in terms of a directly measurable high-level objective, such as a threshold on response time or throughput and uses Bayesian network based classifiers [19] to capture the relationship between these objectives and low-level system metrics. When the high-level objectives are violated, the models determine which low-level metrics are correlated with the violation and which are not; this information can be used to identify likely causes of the performance problem. The assumption is that the Bayesian network classifiers do a good job of capturing patterns of low-level metrics that correlate well with violations of objectives; the approach provides a way of *scoring* its models so it can be determined when this assumption does not hold. Experiments with this approach, both on an experimental testbed and using data from a geographically distributed Enterprise production environment, showed that using a handful of inter-correlated metrics (between 3-8) is often enough to capture between 90-95% of the patterns of normal and abnormal behavior, and generally pointed towards a correct diagnosis/repair action of a performance problem.

A third approach, exemplified by [1], proposes algorithms to reconstruct the *causal* paths followed by transactions through the system, and then identifies path sites corresponding to high time consumption (i.e. possible performance bottlenecks). The assumption is that these causal paths can be reconstructed (in a statistical sense) using time precedence and regularities in the times between the different subtransactions. Note that in this case, there is no consideration of normal or abnormal system behavior. The intent is to provide visibility of the locations where time is spent in the different stages of the transactions. Preliminary results based on different types of traces provide evidence that the algorithms presented in [1] do produce useful and accurate results.

Finally, the work in [38] uses Influence Diagrams to model and tune the parameters for the Berkeley DB embedded database system. Results indicate that the proposed methodology is able to recommend optimal or near optimal tunings for a varied set of workloads including workloads that are not encountered during the model training phase.

Although the above approaches have shown promising initial results, they face some common challenges that are generally not addressed within the scope of the work so far. Even if the most general forms of some of these challenges remain open problems in machine learning, computational learning theory, or data mining, the obstacles may be surmountable for *specific applications* of these approaches to real systems problem with robust engineering solutions.

1. **Model validity:** How can we guarantee that at all times the model being applied is valid, i.e. that it usefully and correctly captures some essential characteristic of the system's operation? This is especially difficult when the behavior of the system being modeled changes dynamically and when both the training data and the "ground truth" for evaluating model accuracy are incomplete or noisy.
2. **Human in the loop:** How do the operators of such systems interpret what is reported by the model? This includes issues such as visualizing results, converting the model's findings to actionable information, dealing with false positives and false negatives,

generating explanations, and enabling the insertion of human expert knowledge and feedback into the models.

3. **Maintaining searchable history of models output** How can we represent the output of the models to enable search of past events and diagnosis/repair actions? This is important so that administrators can leverage past diagnosis efforts, identify quickly recurrent failure modes, among other needs.

3 Challenge 1: A valid model anytime

What should the metrics of "validity" be, given the challenges of determining the ground truth (required to evaluate the model) under the less-than-ideal conditions of a production environment? How do we know that the training data is sufficiently representative of the data seen during production operations—an implicit assumption of most of these approaches? Any realistic long-term resolution of these issues must provide a methodology as well as algorithms and procedures for managing the lifecycle of models, including testing and ensuring their applicability and updating their parameters.

This challenge is not inherent in machine learning itself; indeed, that literature is rife with methods for evaluating and estimating[1] the accuracy of models, and with metrics and scoring functions to compare different models against a dataset [16, 5, 22]. Moreover, statistics textbooks [32] provide algorithms for iterative loops comprising the steps and statistical tests for model evaluation, model diagnosis, and selection of remedial measures to repair the model (if possible). Model diagnosis involves checking whether the assumptions embedded in the models (e.g., linearity of the data, Gaussian noise) correspond to the data at hand; remedial measures may include enhancing the models with additional elements (e.g., metrics), or changing the type of model used (e.g., sets of linear regressions, or nonlinear elements).

Such procedures, while rigorous and well-defined, generally require human intervention to (sometimes visually) check the results of certain steps, adjust parameters and make decisions. The challenge is to automate this process as much as possible by taking advantage of our specific problem domain. A central aspect of this challenge in our domain stems from the complexity and dynamic behavior of the systems we deploy: changes in the system can occur frequently and at unpredictable times. Consequently, the machine learning procedures described above require online implementations so that models can be constantly updated to adapt to the changes in the system. Evidence of this need has been established in [12, 27] with different models and conditions.

Various possible strategies to the model-validity problem might be considered:

1. Build an omniscient model capable of capturing all relevant behaviors. This goal may be unrealistic except in restrictive and benevolent environments. It assumes that at training time we would have access to enough data capturing *all* relevant behaviors.
2. Build a model with an identifiable set of parameters that can adapt to new data. Besides identifying the parameters themselves, this requires mechanisms for identifying the need for adaptation, executing the adaptation (i.e. adjusting the parameters), and data aging.
3. Rely on an ensemble of models. Different models in the ensemble are used in different situations. This requires mechanisms to select which model(s) to use in a given situation, decide when a new model must be added to the ensemble, merge inferences from different models, and discard obsolete models. One example of this approach is described in [41].

There is considerable work in machine learning, computational learning theory (COLT), and data mining addressing these issues (e.g. [9, 29, 4, 20]. The challenge is to adapt these approaches and enrich them with the particulars of our domain.

Another validity-related challenge involves estimating the amount of data required to build accurate models. Despite existing theoretical bounds and much recent progress [14, 25], results for representations such as Bayesian networks don't come easily [21] and researchers often resort to empirical estimation procedures. Although progress on this front has also been made in specific situations (e.g. [41]), we still lack well-engineered general approaches valid in the system domain.

Finally, validation of these models and techniques continues to be a major hurdle. In controlled settings, we may check some of the results by, e.g, injecting specific system conditions and verifying that they are correctly identified/diagnosed by the model. But in production systems, more often than not this "ground truth" will be unavailable, incomplete, or noisy. For example, an operator may suspect that some problem was being manifested in the system during some time period, but be unable to determine conclusively that a particular problem occurred at a particular point in time, or lack sufficient forensic data to reconstruct a problem and diagnose its true root cause (as was reported, e.g., in [7]). To make matters worse, more and more businesses may be willing to provide production data, but either unwilling or unable to provide the ground truth underlying that data,

[1] Since we cannot guarantee that all the pertinent data is available at training time, we can only produce an estimate of the accuracy of a classifier [28].

which is required to objectively measure the success of a method. In other communities, such as computer vision and bioinformatics, standard datasets have been collected and often manually analyzed, providing the means to objectively test and compare different machine learning methods. Such standard datasets are still missing in the system domain. We and our colleagues have called for the creation of an "open source"-like database of real annotated (but sanitized) datasets against which future research in this area could be tested [18], which could do for this line of applied research what the UC Irvine repository did to advance Machine Learning research [6].

4 Challenge 2: The human in the loop

Imagine a system administrator whose responsibility is to execute a triage as soon as system health or performance indicators indicate alarm. Depending on the outcome of the triage, the operator must call the system expert, application expert, network expert, or database expert. Ideally, the administrator would not only offer a justification for the triage and the decision to call a specific expert, but also provide possible explanations for the apparent system misbehavior. Such a scenario, which is quite common in real systems, illustrates that at various levels, humans with different knowledge and levels of expertise would be expected to interact with the models and their inferences. Can the models and their inferences be "interpreted" to generate the justifications and explanations that operators require?

In [12], the choice of Bayesian networks as the basic representation of a model was justified, in part, by the interpretability and modifiability of these models [23, 34]. It is also well known that decision trees can be used to generate "if-then" rules and part of the field of data mining concentrates on these issues [40, 35, 10]. These may provide initial building blocks, but much more research, engineering, and customizations are required to elevate these to the level of usable tools in the systems diagnostics domain.

We take as a given that the problems of false positives and missed detections (false negatives) will always exist. A major e-commerce site has reported false alarm rates in excess of 20% during normal operations. We therefore advocate research directed at minimizing their impact. A first step would be to translate the scores assigned to models during evaluation to a measure of confidence or uncertainty on the recommendations from these models. A second approach is to favor actions that are likely to have a salubrious effect if the alarm is genuine, but have relatively low cost if performed unnecessarily [8]. A framework for combining these ideas may be provided by casting the problem in decision theoretic terms: in this normative approach, the uncertainty of events, the cost/utility of repair actions, and the uncertainty of outcomes are combined to maximize expected utility (minimize expected cost) [34, 15].

In many cases, classifying an alarm as a false positive will still be the prerogative of the human operator. Can we design mechanisms and interfaces so that their expert knowledge can be used to enhance and improve the performance of these models, for example in helping them classify alarms rapidly? Can we also provide mechanisms so that feedback on model performance can be incorporated and used to change these models as appropriate? One strategy is to combine the formal models with other interpretive and diagnostic tools that play to the strengths of humans; for example, [7] presents evidence that combining anomaly detection with visualization allows human operators to exploit their ability for visual pattern recognition to rapidly classify an alarm as a false positive or genuine one. The challenge is to take the data generated by the many sensors, automatically filter out noise, find correlations, and display the information. Another method for using the human operator is known as *active learning*, a method in which the human is queried to provide additional information that would provide the most benefit in reducing false positives and missed detections [30, 13, 39].

5 Challenge 3: Querying the system's past

The third challenge we present concentrates on enabling the creation and management of a searchable history of the sytem's performance. The main benefits of this would be: (a) Similarity-based search for past diagnosis and repairs; (b) identification of recurrent problems[2]; and (c) groupings of problems to enable identification and prioritization.

We concur with Redstone et al. [37] that a first task is constructing a representation that captures the essentials of the system state for characterizing an undesirable (or desirable) observed behavior, and that can be generated in an automatic fashion. We will call this representation a *signature*. Signatures should be amenable to manipulation by computers, such as similarity based retrieval, and to annotation by experts with information regarding previous diagnosis and repairs; these abilities would enable the application of semi-supervised learning methods [31, 33] to improve the retrieval of proven solutions to recurring problems and identification of new problems. Signatures could also be subjected to automated clustering [16] to group similar problems into common "syndromes".

A primarily challenge, then, is to identify suitable sim-

[2] Although we have concentrated on indexing undesirable system states, the same ideas can be used to capture "favorable" states.

ilarity metrics to use in both retrieval and clustering. We attempted to generate signatures from the output of the probabilistic models described in [12, 41] for attributing application level performance problems to specific low-level metrics. During every 5-minute epoch, the models provide a list of system metrics that correlate with a violation of a performance objective, or a list of metrics whose values are abnormal in cases where the system is in compliance with the performance objective. These lists, plus additional information such as degree of correlation to the performance problem and other statistical related measures, are used as the signature. Though our prototype attempts to address the third challenge, in designing it we had to address the first two challenges as well.

Initially we used hand-labeled training data, and induced performance problems to confirm that our signature-generation method displays good similarity retrieval capabilities as well as good clustering properties. The "validity" challenge arose when we applied our technique to a production system. Decisions for the sizes of several windows of data had to be determined would have benefited from principled or well engineered methods for establishing the data needs for accurate models, and how these vary as the input varies. We were encouraged by the fact that we were able to use our signatures to identify all instances of a known problem. This problem took several weeks to identify as being recurrent, and generated over 80 pages' worth of text messages among geographically distributed system operators. Our signatures identified other sets of multiple incidents as potentially belonging to a single "syndrome" (recurring problem), but since the data corresponding to observed performance problems was only partially labeled, we continue to work with the operators to attempt to determine whether these findings and groupings are actually correct.

The "human in the loop" challenge was evident in our struggle to find visualization mechanisms to help operators compare different signatures. In addition, we still lack a systematic way to incorporate operators' expertise back into our methods. These problems are further confounded by the fact that responsibility for different tiers of our production system (application server, database, etc.) spans organizational boundaries across which there are differing techniques for data collection, troubleshooting, and alarm handling.

6 Conclusions

Recent research has demonstrated that it is possible to automatically induce models from raw data collected from a running networked system, and that these models can indeed transform raw data into useful information for many tasks related to performance debugging and isolation, anomaly detection, detecting and localizing non-failstop failures, among others. We are excited by the potential of these approaches in increasing the efficacy and efficiency of the management of complex IT systems.

With IT budgets dominated by human operator costs, the potential benefits would be significant even if these techniques only served to increase the effectiveness of less-experienced operators. We believe, however, that even experts will benefit from being able to quantify their intuitions about correlations, breaking points, and patterns of behavior. In addition, the possibilities of exploring the data efficiently will provide tools for testing new hypotheses and "what-if" scenarios.

We do not, of course, advocate statistical, probabilistic modeling, and pattern recognition techniques as the solution to all "self-*" problems. Beyond the well-known limitations of the benefit of automation and the problem of "automation irony" [36], the essence of the proposed research agenda is to understand the particular limitations of statistical approaches as applied to system problem detection, localization, and ultimately diagnosis. In order to understand these limits, we must identify the fundamental challenges that will be faced by any work in this area. We have attempted to formulate three such challenges and show how they arise in real problem instances. With the availability of high-quality open-source implementations of statistical induction and pattern recognition algorithms [3, 24, 40] and increasing interest in the integration of measurement frameworks with system middleware, now is the time to vigorously pursue this line of research and identify the limits and benefits of these approaches.

7 Acknowledgments

We thank P. Bodík, J. Chase, T. Kelly, E. Kıcıman, J. Mogul, D. Patterson, J. Symons, and M. Verber for discussions that contributed to these ideas, and the anonymous referees whose comments greatly improved the paper.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *19th Symposium on Operating Systems Principles*, 2003.

[2] P. Barham, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modeling. In *6th Symp. on Operating Systems Design and Implementation* , 2004.

[3] Bayesian network classifier toolbox. `http://jbnc.sourceforge.net/`.

[4] J. Binder, D. Koller, S. J. Russell, and K. Kanazawa. Adaptive probabilistic networks with hidden variables. *Machine Learning*, 29, 1997.

[5] C. Bishop. *Neural Networks for Pattern Recognition*. Oxford, 1995.

[6] C. Blake and C. Merz. UCI repository of machine learning databases, 1998.

[7] P. Bodík, G. Friedman, L. Biewald, H. Levine, G. Candea, A. Fox, M. I. Jordan, D. Patterson, K. Patel, G. Tolle, and J. Hui. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *International Conference on Autonomic Computing*, 2005.

[8] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. A microrebootable system – design, implementation, and evaluation. In *6th Symposium on Operating Systems Design and Implementation*, 2004.

[9] N. Cesa-Bianchi, Y. Freund, D. P. Helmbold, and M. Warmuth. On-line prediction and conversion strategies. In *Computational Learning Theory: Eurocolt*, 1993.

[10] M. Chen, A. Zheng, J. Lloyd, M. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *International Conference on Autonomic Computing*, 2004.

[11] T. Chou. *The End of Software*. Sams Publishing, Indianapolis, IN, 2005.

[12] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. Chase. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *6th Symp. on Operating Systems Design and Implementation*, 2004.

[13] D. A. Cohn, Z. Ghahramani, and M. I. Jordan. Active learning with statistical models. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, 7, 1995.

[14] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2000.

[15] M. DeGroot. *Optimal Statistical Decisions*. McGraw-Hill, 1970.

[16] R. Duda and P. Hart. *Pattern Classification and Scene Analysis*. John Wiley and Sons, 1973.

[17] A. Fox and D. Patterson. Self-repairing computers. *Scientific American*, June 2003.

[18] A. Fox, D. A. Patterson, and M. I. Jordan. Reliable adaptive distributed systems (research proposal). National Science Foundation Award, June 2005.

[19] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29, 1997.

[20] N. Friedman and M. Goldszmidt. Sequential update of Bayesian network structure. In *International Conference on Uncertainty in Artificial Intelligence*, 1997.

[21] N. Friedman and Z. Yakhini. On the sample complexity of learning Bayesian network. In *International Conference on Uncertainty in Artificial Intelligence*, 1996.

[22] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning*. Springer, 2001.

[23] D. Heckerman, D. Geiger, and D. Chickering. Learning Bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, 20, 1995.

[24] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3), 1996.

[25] M. Kearns and U. Vazirani. *An Introduction to Computational Learning Theory*. MIT press, 1994.

[26] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[27] E. Kıcıman and A. Fox. Detecting application-level failures in component-based internet services. Submitted, Sept. 2004.

[28] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International Joint Conference on Artificial Intelligence*, 1995.

[29] T. Lane and C. E. Brodley. Approaches to online learning and concept drift for user identification in computer security. In *International Conference on Knowledge Discovery and Data Mining*, 1998.

[30] D. MacKay. Information-based objective functions for active data selection. *Neural Computation*, 4(4), 1992.

[31] T. Mitchell. The role of unlabeled data in supervised learning. In *Proc. of International Colloquium on Cognitive Science*, 1999.

[32] J. Neter, M. Kutner, C. Nachtshein, and W. Wasserman. *Applied Linear Statistical Models*. McGraw-Hill, 1996.

[33] K. Nigam, A. McCallum, S. Thrun, and T. Mitchell. Text classification from labeled and unlabeled documents using EM. *Machine Learning*, 39, 2000.

[34] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[35] J. Quinlan. *C4.5 Programs for machine learning*. Morgan Kaufmann, 1993.

[36] J. Reason. *Managing the Risks of Organizational Accidents*. Ashgate Publishing Co., 1997.

[37] J. Redstone, M. Swift, and B. Bershad. Using computers to diagnose computer problems. In *Proc. 9th Workshop on Hot Topics in Operating Systems*, 2003.

[38] D. Sullivan. *Using probabilistic reasoning to automate software tuning*. PhD thesis, Harvard University, 2003.

[39] S. Tong and D. Koller. Support vector machine active learning with applications to text classification. In *17th International Conference on Machine Learning*, 2000.

[40] I. Witten and E. Frank. *Data Mining: Practical machine learning tools with Java implementations*. Morgan Kaufmann, 2000.

[41] S. Zhang, I. Cohen, M. Goldszmidt, J. Symons, and A. Fox. Ensembles of models for automated diagnosis of system performance problems. In *Intl. Conf. on Dependable Systems and Networks*, 2005.

Making system configuration more declarative

John DeTreville
Microsoft Research
johndetr@microsoft.com

Abstract

System administration can be difficult and painstaking work, yet individual users must typically administer their own personal systems. These personal systems are therefore likely to be misconfigured, undependable, brittle, and insecure, which restricts their wider adoption. Because updating the configuration of today's systems involve imperative updates in place, a system's correctness ultimately depends on the correctness of every install and uninstall it has ever performed; because these updates are local in scope, there is no easy way to specify or check desired properties for the whole system. We present a more checkable declarative approach to system configuration that should improve system integrity and make systems more dependable. As in the earlier Vesta system, we define a *system model* as a function that we can apply to a collection of system parameters to produce a statically typed, fully configured *system instance*; models can reference and thereby incorporate *submodels*, including submodels exported by each program in the system. We further check each system instance against established *system policies* that can express a variety of additional *ad hoc rules* defining which system instances are acceptable. Some system policies are expressible using additional type rules, while others must operate outside the type system. A preliminary design and implementation of this approach are under way for the Singularity OS, and we hope to specify and check a number of ad hoc system properties for Singularity-based personal systems.

1. Terminology and introduction

Programmers write programs; *administrators* configure these programs into systems; *users* apply these systems to their tasks.

Some individuals combine these roles, but most do not. For instance, some expert users are also expert programmers but most are not; most users rely instead on the large available body of general-purpose programs written by others.

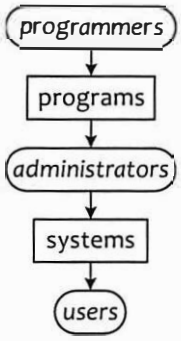


Figure 1. Programmers, administrators, and users.

Similarly, some expert users are also expert administrators but most are not. System administration can be difficult and painstaking work; programs in a system can interact in unexpected ways, and installing one program can very readily break another. Unfortunately, the users of personal systems must typically administer their own systems, and we believe that this creates barriers to the wider adoption of new personal systems.

As a result, a great many personal systems are poorly configured and poorly maintained. They do not work well. They are undependable. They are brittle. They are insecure. How might we do better?

2. What can go wrong?

Let's consider a (very) simple example. The user, acting as *de facto* administrator, chooses four programs—photo editor E, camera driver C, printer driver P, and kernel K—and configures them into the system shown in Figure 2. What can go wrong?

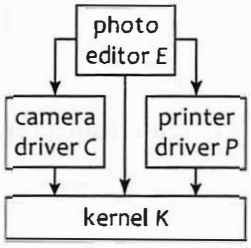


Figure 2. One system configuration.

- The user can choose programs that simply do not work together—at all. Printer driver P may require a formatting language that photo editor E cannot produce, or produces incorrectly. If there are multiple versions of P and E, the user can choose a bad pair.
- The user can misconfigure the programs, causing them not to work together. Misconfiguring UTF-8 support in kernel K, let's say, might change its semantics enough to break its clients.

Most existing configuration tools are imperative in nature. The system configuration exists as mutable state in the file system, in the Windows registry, *etc.*, and the *de jure* or *de facto* administrator updates the configuration in place by installing and uninstalling programs. A system's correctness therefore depends on the correctness and the appropriateness of each install and uninstall the system has ever performed, as well as their exact order.

- Real systems evolve, and even an initially good system can become misconfigured over time. Many configuration settings are shared or global, meaning that local updates can far too readily create global problems.

The result is that configuration management in current systems falls short in several ways.

- System configurations are *brittle, imperative,* and *history-dependent,* especially since our tools for managing configurations can deal only with *local* constraints. Let's imagine that installing program C or P must reconfigure K. If we install C, then P, is K still correctly configured for C? If we uninstall C, is K still correctly configured for P?
- System configurations are *imprecise* and overly *dynamic.* When a program uses another program—perhaps as a library, perhaps as a service, perhaps otherwise—the system can choose the other program in some arbitrary fashion at runtime, such that we cannot check the combination statically.
- System configurations are *insecure.* When a system boots, it runs whatever system it finds on its hard drive, with no opportunity for enforcing an end-to-end check.

3. What has been tried?

There are several existing approaches for improving the task of system configuration, each with its own shortcomings.

3.1. Central administration

Central administration works well in many enterprise environments, where expert professional administrators can create some small number of standard configurations. These central administrators can choose, customize, and configure programs to work well together, and maintain the resulting configurations over time.

Central administration seems much less suitable for personal systems. Home systems, for example, are quite varied and quite frequently reconfigured. As other personal systems, such as mobile phones, become more like home systems, they also become less amenable to central administration. We therefore should not expect central administration to work well for personal systems.

We also propose that, all else being equal, users' interests are best served when they can choose their own programs, even if they need not.

3.2. Closed systems

A similar approach is the closed system, where a system's programs all come from a single supplier or integrator. Closed systems are common in the world of consumer electronics, where the manufacturer delivers and upgrades a typical system's firmware monolithically.

Most existing personal systems are not closed, except for the very simplest, and closed systems seem less and less suited over time to satisfy the ever-expanding needs and expectations of individual users. Simple closed systems cannot necessarily scale to serve complex, varied environments.

3.3. Stronger isolation

Can we factor our open systems into some number of closed programs that do not interact? Each program might execute in a separate virtual machine or virtual environment without interfering with the others.

No. Real programs interoperate with each other. Program C copies photos from a digital camera; E edits them; P prints them. Reducing extraneous interaction between programs can reduce interference, of course, but real programs will always interact. We must allow users to choose their programs independently, even though these programs can and will interact.

3.4. Stronger interfaces

Many systems let programs interact only across strongly typed interfaces. Strong static typing can eliminate many mismatches and misconfigurations, but it is not a panacea; a program A can work perfectly well with B but not at all with the identically typed B′, and then again with B″.

Some bad configurations won't type-check, but many more will have subtler problems. We need solutions that are more powerful than strongly typed interfaces as they currently exist.

3.5. Better programs

If program P works with K but not with K′, doesn't that mean that K satisfies its contract and K′ does not?—or that P is somehow depending on unspecified behavior? Can't we just write P, K, and K′ correctly in the first place?

No, in general, we can't. We believe that our programs will continue to have bugs, and our interfaces will continue to elide important information. We will continue to integrate programs from different programmers with different assumptions, and we will continue to discover their interfaces and requirements experimentally. In short, we will continue to integrate imperfect programs for the near future, and perhaps longer.

3.6. Smarter installers

Some installers can explicitly model programs' dependence on each other, eliminating some misconfigura-

tions; examples include Windows Installer [10] and the Debian package management system [1]. If P and K′ do not work together, installing P might also upgrade K′ to K″, while upgrading K to K′ should perhaps fail if P is already installed.

Existing installers of this sort typically can check system configurations for local consistency but not for global consistency. They can avoid some misconfigurations but not others.

3.8. Smarter users

Many people argue that users should better understand the internal workings of their systems, and that administering their systems helps them learn. If users learn enough about how their systems work, the argument goes, then they can configure their systems as they see fit. Conversely, if users don't really understand their systems, then they get what they deserve.

We argue the opposite. Eliminating the need for users to administer their own systems should be as beneficial as eliminating the need for them to develop their own programs. Most users—who are neither expert programmers nor expert administrators—are better off when others can perform these tasks for them.

4. Declarative configuration

We propose a *declarative* approach to system configuration that addresses many of these problems. Our proposal derives from the earlier Vesta software configuration system [4] [5], which itself derived from the Cedar System Modeller [9].

- We compose a declarative *system model* that completely and precisely specifies the system as a whole.
- Evaluating the system model, as applied to the *system parameters*, produces a complete, fully configured *system instance*.
- Extending the Vesta approach, we can further check each system instance against established *system policies* that can express a variety of ad hoc rules that define which system instances are acceptable.

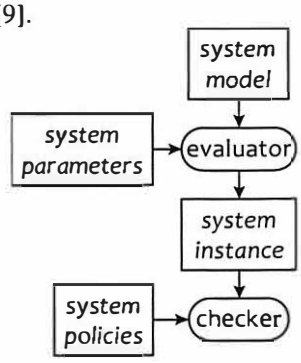


Figure 3. Models, parameters, instances, and policies.

We argue that this declarative approach to system configuration can improve the integrity and thus the dependability of personal systems. (Other analyses of the problems of system administration have also focused on mutable configuration state [6]; our declarative approach can eliminate much of this mutable state.) A preliminary design and implementation of this approach are under way for Singularity, a new research OS intended to support the construction of dependable systems [7] [8].

4.1. Models

Models are hierarchical. The system model can reference—and thus incorporate—any number of submodels, usually including one for each component program, and these submodels can themselves be hierarchical. Programmers, publishers, and remote administrators can write these submodels, while the local administrator composes them into the local system model. Our goal when writing system models and submodels is to express rules for how we can correctly compose the various programs into systems. Our hope is that system models can be easy to compose from their submodels.

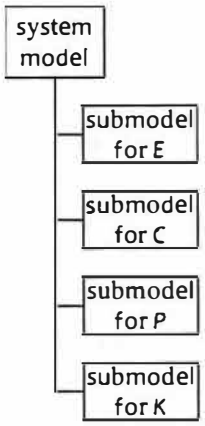


Figure 4. A system model and its submodels.

In our example, the system model incorporates submodels for programs E, C, P, and K. We apply each program model to its appropriate parameters to produce a program instance, and we compose these program instances into a fully configured system instance.

Let's consider kernel K from Figure 2. (We present these examples in the functional language Haskell [11] [3], although our implementation for Singularity may not itself use Haskell.) A program instance exports some number of values. The kernel instance in our example (examples are partially elided in this paper) exports the kernel's identity (a secure hash of type `Hash`) and a *reboot* operation (of type `KReboot`).

```
> data K = K Hash KReboot
```

The function `kModel` is our kernel model. It takes no parameters, and returns a kernel instance of type `K`.

```
> kModel ()
>   = K (Sha256 "b6f8…2ab7") doKReboot
```

(This partially elided hash identifies the binary for kernel K. A more realistic example might return different hashes depending on its parameters.) Here, `doKReboot` implements the *reboot* operation for kernel K.

We define the types `C`, `P`, and `E`, and the functions `cModel`, `pModel`, and `eModel`, similarly.

Finally, the data type `System` represents the system instance; it exports its secure hash (of type `Hash`) and a

run operation (of type SRun), along with its component program instances.

```
> data System = System Hash SRun K C P E
```

The system model systemModel is a function that takes the four program instances (of types K, C, P, and E) and returns a system instance (of type System).

```
> systemModel k c p e
>   = System
>       (bind [hash k, hash c,
>              hash p, hash e])
>       doSRun k c p e
```

The function bind links a number of programs, identified in this example by their secure hashes, and returns the secure hash of the result; doSRun is a function that implements the system's *run* operation.

4.2. Evaluation

Applying a model to its parameters evaluates to an instance. We produce instances k, c, p, e, and system of types K, C, P, E, and System.

```
> k = kModel ()
> c = cModel k
> p = pModel True k
> e = eModel True k c p
> system = systemModel k c p e
```

(Here, pModel and eModel each take one extra Bool parameter.) The resulting value system is the fully configured system instance, which exports k, c, p, and e.

Model evaluation has no side effects, and applying the same system model to the same parameters always produces the same system instance. We can produce a new system instance from an updated model, or from an old model with updated parameters, but we always produce it functionally, and not as a local update to the current system instance on the current machine.

The functional nature of model evaluation is convenient for system administrators, especially for the administrators of distributed systems. For example, it lets us produce system instances on systems other than the ones on which they will run. It might be much simpler to construct a new system model for a light switch on a personal computer or some similarly powerful system than on the light switch itself.

4.3. Type checking and subtyping

Not only are our system instances and program instances values, they are also statically typed and statically checkable. Our models, *etc.*, are also statically checkable.

In our example, a system instance of type System must contain a kernel instance of type K, and a system instance will not type-check if another type is used. When this is too constraining—perhaps we would like to use a kernel of type K' that also exports a *shutdown* operation, so that K' <: K—we can use *subtyping* to express looser rules. Here, we redefine System to include any kernel type k that exports at least a *reboot* operation, as defined by the HasKReboot type class. (Belonging to a Haskell type class is like implementing a C# or Java interface.)

```
> data System
>   = forall k .
>       HasKReboot k
>       => System Hash SRun k C P E
```

(Here, k is an existentially quantified type variable.) We also declare our own type K to belong to the type class HasKReboot. We can make similar changes elsewhere in our example to take further advantage of subtyping.

4.4. Installation

Installing a new system instance involves three steps.

1) We make the new system instance available on the local machine or across the network.
2) We make the new system instance *current* by setting the local machine to boot only from that instance, as specified by the instance's secure hash.
3) We atomically reboot the local machine.

(We expect that we can eliminate the reboot in many cases.) More than one system instance can be available at once—and they can share common structure—but only one can be current at a time.

We provide no way for an installer or an administrator to modify a system instance in place. (Such imperative edits are brittle because the correctness of the system depends on the correctness of all of these edits over its lifetime.) Since our system instances are immutable, we can refer to them by their secure hashes.

Because of our all-at-once approach to installation, the order in which system instances are produced and installed does not matter, and no sequence of installs and uninstalls can result in a badly formed system instance.

4.5. Runtime

As shown in Section 0, one program instance can reference others; in our example, c is a C with a field named cK that is the instance of kernel K for the system. When a system instance boots, the hardware can check that it is the current system instance, and refuse to proceed if it is not.

As stated in Section 4.1, system instances and program instances export values, which can reference other instances; in our example, a P might export two values: a Bool and a K.

```
> data P = P Bool K
```

We let each program read its own program instance at runtime, allowing it to read and act upon the values that it exports. In this case, the `Bool` might have been a parameter to `pModel`, intended to control P's execution.

4.6. Policies

Configuring real systems requires one to know a great many *ad hoc* rules. One rule might be that program P is known to work with K and not K′; another might be that P has not been tested against K″ but that it ought to work anyway—assuming that its `Bool` parameter was `True`. We call these *ad hoc rules*, and we argue that ad hoc rules account for much of the difficulty of real system configuration. Our *system policies* therefore provide a way to express a variety of ad hoc rules that can further constrain the acceptable structure of the system. We need these ad hoc rules because our programs are not perfect, and because their most interesting properties are often not discovered until after they are written and deployed. System administration is often messy and unstructured, and system policies let us express these ad hoc rules.

We can implement many of these system policies using additional type rules. Imagine that program E requires a kernel that supports UTF-8. We can encode this policy by saying that its kernel must belong to the `Utf8Support` type class (perhaps among others).

```
> data E
>   = forall k .
>       (Utf8Support k, HasKReboot k)
>       => E Hash ERun k C P
```

Each known kernel type can then be listed as belonging to the `Utf8Support` type class or not. When new determinations are made—perhaps a new kernel is published, or perhaps an old kernel is found not to support UTF-8 to our satisfaction—we can import new definitions and act on them. While we must make these annotations manually, we can check them automatically.

For other policies, when type rules are not so directly applicable—for example, if there is a policy that the system must fit in less than a megabyte of RAM—an ad hoc checker can traverse the system instance and check it against the desired policy.

Some system policies can be authored by the local system administrator, while may accompany programs from elsewhere, and yet others may come from third parties. The local system administrator can choose to adopt these imported policies or not.

If a system instance does not conform to the governing policies, the evaluator will not produce it and we cannot use it; we must change the model or its parameters for it to become acceptable.

4.7. Attribution

Another ad hoc policy—for example—might be that the local system must provide a good French-language UI. We might redefine a `System`'s program instances as belonging to the type class `Français`.

```
> data System
>   = forall k c p e .
>       (Français k, Français c,
>        Français p, Français e)
>       => System Hash SRun k c p e
```

We can then define our program instances—E, for example—as belonging to `Français`.

```
> instance Français E
```

But who writes this instance definition? What is a "good" French-language UI? Who gets to decide? And how might we check so ill-defined a policy?

Our rule is that the local system administrator makes such decisions, and a local system instance belongs to the type class `Français` if and only if the local administrator says so. The local administrator can of course choose to defer to the program's publisher when appropriate, or to other authorities—perhaps to the Académie Française, which could publish its own policies. The earlier Binder security language provides mechanisms for attribution and deferral ("delegation") in a distributed environment [2], and we would expect that its mechanisms should be useful here too.

Another policy might more realistically insist that the system's component programs not have been named in US-CERT security alerts, as defined by US-CERT [13]. Ongoing security alerts arriving at a system could cause the system no longer to meet its policy, perhaps notifying an administrator.

4.8. Extensions

We hope to specify and check a variety of system properties using the approaches described here, and we hope we can extend these approaches to extend the properties we can specify and check.

Our current system instances are static, but we plan also to support *dynamic instances* to model the system's runtime state. A program will be able to read its own dynamic program instance, referencing other dynamic program instances; this could provide a foundation for easily configurable inter-program communications.

Real system state can seem quite complex. This paper was written on a system with 216,141 files and 17,663 folders, but many of its 233,804 ACLs are little more than accidents of history. While there is little chance that these ACLs are all correct—whatever that might mean!—there may be some greater chance that

we can write concise policies that can check the ACLs. Perhaps such system state is not as complex as it seems!

Expressing our ad hoc policies as type rules requires a powerful and flexible underlying type system. While Haskell has an excellent type system, one can certainly imagine improvements. Extensible record types were used in Vesta and may be useful here as well. Unlike Vesta, we expect that we can check these types statically.

Our current approach to system security is restricted to ensuring system integrity. We hope also to address confidentiality in future extensions.

In our current design we avoid the inviting possibility of fixing system configuration problems automatically as they are detected, such as by substituting a better version of a kernel, since doing so currently seems much more error-prone than relying on humans to fix these problems. We expect to revisit this decision.

5. Feasibility

Is this approach to system configuration feasible? The only sure way to tell for sure is to build it and use it, but we have some intuitions suggesting that it could work.

While earlier efforts at declarative configuration, like Vesta and the CML2 kernel configuration language [12], have not been widely adopted, they were targeted at programmers who already used and understood the existing configuration tools, and who were therefore disinclined to switch. This may not be a problem with personal systems, where the need for new tools for users and administrators should be more obvious.

Our system models and system policies may be too complex and too difficult to get right. We argue only that they will be smaller, simpler, and more precise than the system instances they produce and check.

Since many people will write submodels, we must create standards to allow their interoperation, and ensure that malicious submodels cannot hijack a system. Our current understanding of these problems is inadequate, but it should improve with further experience.

While we have certainly not eliminated the need for system administration, we believe that we have reduced the work involved. A sufficient reduction should allow us to outsource the remaining administration tasks, including detecting, diagnosing, and repairing any problems that otherwise elude us.

Finally, we note that we have based this work in its entirety on the assumption that the complexity of system configuration limits the use and acceptance of personal systems. We have no quantitative evidence to support this assumption, although we do have a growing collection of supporting anecdotes.

References

[1] J. H. M. Dassen, Chuck Stickelman, Susan G. Kleinmann, Sven Rudolph, and Josip Rodin. *The Debian GNU/Linux FAQ chapter 6—Basics of the Debian package management system.* February 2003.

[2] John DeTreville, "Binder, a logic-based security language." *Proceedings of the 21st IEEE Symposium on Security and Privacy,* Oakland, California, pp. 105–113, May 2002.

[3] The GHC Team. *The glorious Glasgow Haskell compilation system user's guide.* Version 6.4, March 2005.

[4] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. "The Vesta approach to software configuration management." Compaq Systems Research Center Research Report 168, March 2001.

[5] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. "The Vesta software configuration management system." Compaq Systems Research Center Research Report 177, January 2002.

[6] David A. Holland, William Josephson, Kostas Magoutis, Margo I. Seltzer, Christopher A. Stein, and Ada Lin. "Research issues in no-futz computing." *Proceedings of the 8th Workshop on Hot Topics in Operating Systems,* pp. 106–112, Schoss Elmau, Germany, May 2001.

[7] Galen C. Hunt and James R. Larus. "Singularity design motivation." Microsoft Research Technical Report MSR-TR-2004-105, November 2004.

[8] Galen C. Hunt, James R. Larus, David Tarditi, and Ted Wobber. "Broad new OS research: Challenges and opportunities." *Proceedings of the 10th Workshop on Hot Topics in Operating Systems,* Santa Fe, New Mexico, June 2005.

[9] Butler W. Lampson and Eric E. Schmidt. "Organizing software in a distributed environment." ACM SIGPLAN Notices 18, 6 (June 1983), pp. 1–13.

[10] Microsoft Corporation. *Microsoft Platform SDK: Windows Installer*: November 2004.

[11] Simon Peyton Jones, editor. *Haskell 98 language and libraries: The revised report.* Cambridge University Press, 2003.

[12] Eric S. Raymond. *The CML2 resources page.* February 2002.

[13] United States Computer Emergency Readiness Team (US-CERT), National Cyber Security Division, Department of Homeland Security. *Technical cyber security alerts,* 2005.

Reducing the Cost of IT Operations—Is Automation Always the Answer?

Aaron B. Brown and Joseph L. Hellerstein
IBM Thomas J. Watson Research Center
Hawthorne, New York, 10532
{abbrown,hellers}@us.ibm.com

Abstract

The high cost of IT operations has led to an intense focus on the automation of processes for IT service delivery. We take the heretical position that automation does not necessarily reduce the cost of operations since: (1) additional effort is required to deploy and maintain the automation infrastructure; (2) using the automation infrastructure requires the development of structured inputs that have up-front costs for design, implementation, and testing that are not required for a manual process; and (3) detecting and recovering from errors in an automated process is considerably more complicated than for a manual process. Our studies of several data centers suggest that the up-front costs mentioned in (2) are of particular concern since many processes have a limited lifetime (e.g., 25% of the packages constructed for software distribution were installed on fewer than 15 servers). We describe a process-based methodology for analyzing the benefits and costs of automation, and hence for determining if automation will indeed reduce the cost of IT operations. Our analysis provides a quantitative framework that captures several traditional rules of thumb: that automating a process is beneficial if the process has a sufficiently long lifetime, if it is relatively easy to automate (i.e., can readily be generalized from a manual process), and if there is a large cost reduction (or leverage) provided by each automated execution of the process compared to a manual invocation.

1 Introduction

The cost of information technology (IT) operations dwarfs the cost of hardware and software, often accounting for 50% to 80% of IT budgets [8, 4, 16]. IBM, HP, and others have announced initiatives to address this problem. Heeding the call in the 7th HotOS for "futz-free" systems, academics have tackled the problem as well, focusing in particular on error recovery and problem determination. All of these initiatives have a common message: *salvation through automation*. This message has appeal since automation provides a way to reduce labor costs and error rates as well as increase the uniformity with which IT operations are performed.

After working with corporate customers, service delivery personnel, and product development groups, we have come to question the widely held belief that automation of IT systems always reduces costs. In fact, our claim is that automation can *increase* cost if it is applied without a holistic view of the processes used to deliver IT services. This conclusion derives from the hidden costs of automation, costs that become apparent when automation is viewed holistically. While automation may reduce the cost of certain operational processes, it increases other costs, such as those for maintaining the automation infrastructure, adapting inputs to structured formats required by automation, and handling automation failures. When these extra costs outweigh the benefits of automation, we have a situation described by human factors experts as an *irony of automation*—a case where automation intended to reduce cost has ironically ended up increasing it [1].

To prevent these ironies of automation, we must take a holistic view when adding automation to an IT system. This requires a technique for methodically exposing the hidden costs of automation, and an analysis that weighs these costs against the benefits of automation. The approach proposed in this paper is based on explicit representations of IT operational processes and the changes to those processes induced by automation. We illustrate our process-based approach using a running example of automated software distribution. We draw on data collected from several real data centers to help illuminate the impact of automation and the corresponding costs, and to give an example of how a cost-benefit analysis can be used to determine when automation should and should not be applied. Finally, we broaden our analysis into a general discussion of the trade-offs between manual and automated processes and offer guidance on the best ways to apply automation.

2 Hidden Costs of Automation

We begin our discussion of the hidden costs of automation by laying out a methodical approach to exposing them. Throughout, we use software distribution to server machines as a running example since the proper management of server software is a critical part of operating a data center. Our discussion applies to software package management on centrally-administered collections of desktop machines as well. Software distribution in-

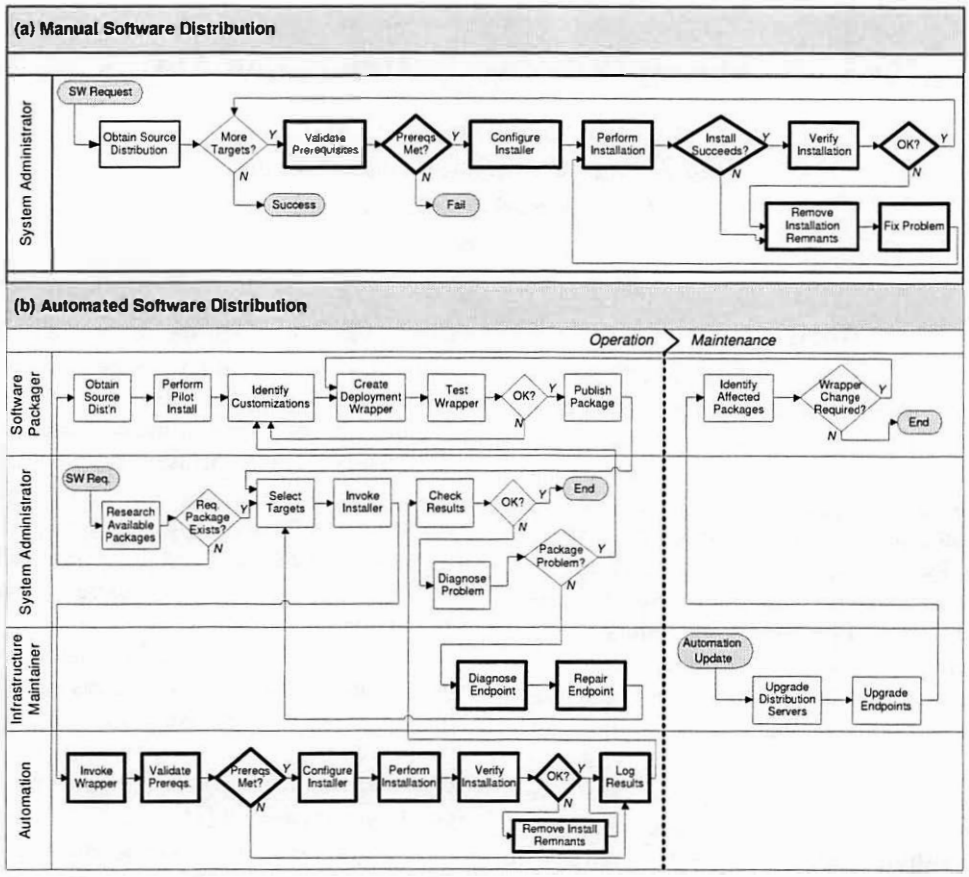


Figure 1: ***Manual and automated processes for software distribution.*** *Boxes with heavy lines indicate process steps that contribute to variable (per-target) costs, as described in Section 3.*

volves the selection of software components and their installation on target machines. We use the term "package" to refer to the collection of software resources to install and the step-by-step procedure (process) by which this is done.

Our approach is based on the explicit representation of the *processes* followed by system administrators (SAs). These processes may be formal, *e.g.* derived from ITIL best practices [13], or informal, representing the ad-hoc methods used in practice. Regardless of their source, the first step is to document the processes as they exist before automation. Our approach accomplishes this with "swim-lane" diagrams—annotated flowcharts that allocate process activities across *roles* (represented as rows) and *phases* (represented as columns). Roles are typically performed by people (and can be shared or consolidated); we include automation as its own role to reflect activities that have been handed over to an automated system.

Figure 1(a) shows the "swim-lane" representation for the manual version of our example software distribution process. In the data centers we studied, the SA responds to a request to distribute software as follows: (1) the SA obtains the necessary software resources; (2) for each server, the SA repeatedly does the following—(2a) checks prerequisites such as the operating system release level, memory requirements, and dependencies on other packages; (2b) configures the installer, which requires that the SA determine the values of various parameters such as the server's IP address and features to be installed; and (2c) performs the install, verifies the result, and handles error conditions that arise. While Figure 1(a) abstracts heavily to illustrate similarities between software installs, we underscore that a particular software install process has many steps and checks that typically make it quite different from other seemingly similar software installs (e.g., which files are copied to what directories, pre-requisites, and the setting of configuration parameters).

Now suppose that we automate the process in Figure 1(a) so as to reduce the work done by the SA. That is, in the normal case, the SA selects a software package, and the software distribution infrastructure handles the other parts of the process flow in Figure 1(a). Have we simplified IT operations?

No. In fact, we may have made IT operations *more*

complicated. To understand why, we turn to our process-driven analysis, and update our process diagram with the changes introduced by the automation. In the software distribution case, the first update is simple: we move the automated parts of Figure 1(a) from the System Administrator role to a new Automation role. But that change is not the only impact of the automation. For one thing, the automation infrastructure is another software system that must itself be installed and maintained. (For simplicity, we assume throughout that the automation infrastructure has already been installed, but we do consider the need for periodic updates and maintenance.) Next, using the automated infrastructure requires that information be provided in a structured form. We use the term *software package* to refer to these structured inputs. These inputs are typically expressed in a formal structure, which means that their creation requires extra effort for package design, implementation, and testing. Last, when errors occur in the automated case, they happen on a much larger scale than for a manual approach, and hence additional processes and tools are required to recover from them.

These other impacts manifest as additional process changes, namely extra roles and extra operational processes to handle the additional tasks and activities induced by the automation. Figure 1(b) illustrates the end result for our software distribution example. We see that the automation (the bottom row) has a flow almost identical to that in Figure 1(a). However, additional roles are added for care and feeding of the automation. The responsibility of the System Administrator becomes the selection of the software package, the invocation of the automation, and responding to errors that arise. Since packages must be constructed according to the requirements of the automation, there is a new role of Software Packager. The responsibility of the packager is to generalize what the System Administrator does in the manual process so that it can be automated. There is also a role for an Infrastructure Maintainer who handles operational issues related the software distribution system (e.g., ensuring that distribution agents are running on endpoints) and the maintenance of the automation infrastructure.

From inspection, it is apparent that the collection of processes in Figure 1(b) is much more complicated than the single process in Figure 1(a). Clearly, such additional complexity is unjustified if we are installing a single package on a single server. This raises the following question—at what point does automation stop adding cost and instead start reducing cost?

3 To Automate or Not To Automate

To answer this question, we first characterize activities within a process by whether they are used for setup (the outer part of a loop) or per-instance (the inner part of the loop). Boxes with heavy outlines in Figure 1 indicate the per-instance activities. Note that in Figure 1(b), most of the per-instance activities are done by the automation. We refer to the setup or up-front costs as **fixed costs**, and the per-instance cost as **variable costs**.

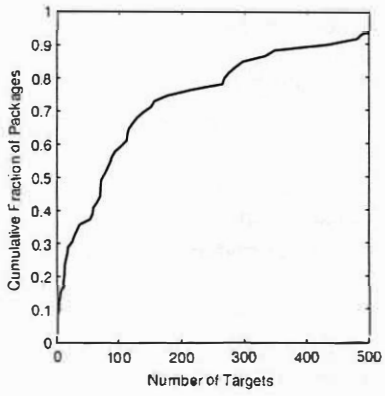


Figure 2: *Cumulative distribution of the number of targets (servers) on which a software package is installed over its lifetime in several data centers. A larger number of packages are installed on only a small number of targets.*

A rule-of-thumb for answering the question above is that automation is desirable if the variable cost of the automated process is smaller than the variable cost of the manual process. *But this is wrong.*

One reason why this is wrong is that we cannot ignore fixed costs for automating processes with a limited lifetime. IT operations has many examples of such limited lifetime processes. Indeed, experience with trying to capture processes in "correlation rules" used to respond to events (e.g., [10, 5]) has shown that rules (and hence processes) change frequently because of changes in data center policies and endpoint characteristics.

Our running example of software distribution is another illustration of limited lifetime processes. As indicated before, a software package describes a process for a specific install; it is only useful as long as that install and its target configuration remain current. The fixed cost of building a package must be amortized across the number of targets to which it is distributed over its lifetime. Figure 2 plots the cumulative fraction of the number of targets of a software package based on data collected from a several data centers. We see that a large fraction of the packages are distributed to a small number of targets, with 25% of the packages going to fewer than 15 targets over their lifetimes.

There is a second reason why the focus on variable costs is not sufficient. It is because the focus is on the variable costs of *successful* results. By considering the complete view of the automated processes in Figure 1(b), we see that more sophistication and people are required to address error recovery for automated soft-

ware distribution than for the manual process. Using the same data from which Figure 2 is extracted, we determined that 19% of the requested installs result in failure. Furthermore, at least 7% of the installs fail due to issues related to configuration of the automation infrastructure, a consideration that does not exist if a manual process is used. This back-of-the envelope analysis underscores the importance of considering the entire set of process changes that occur when automation is deployed, particularly the extra operational processes created to handle automation failures. It also suggests the need for a quantitative model to determine when to automate a process.

Motivated by our software distribution example, we have developed a simple version of such a model. Let C_f^m be the fixed cost for the manual process and C_v^m be its variable cost. We use N to denote lifetime of the process (e.g., a package is distributed to N targets). Then, the total cost of the manual process is

$$C^m = C_f^m + NC_v^m$$

Similarly, there are fixed and variable costs for the automated process. However, we observe from Figure 1(a) and Figure 1(b) that the fixed costs of the manual process are included in the fixed cost of the automated process. We use C_f^a to denote the *additional* fixed costs required by the automated process, and we use C_v^a to denote the variable cost of the automated process. Then, the total cost of the automated process is

$$C^a = C_f^m + C_f^a + NC_v^a$$

The costs can be obtained through billing records, as we have done at IBM. N depends on the packages being distributed and the configuration of potential targets.

We can make some qualitative statements about these costs. In general, we expect that $C_v^m > C_v^a$; otherwise there is little point in considering automation. Also, we expect that $C_v^m \leq C_f^a$ since careful design and testing are required to build automation, which requires performing the manual process one or more times. Substituting into the above equations and solving for N, we can find the crossover point where automation becomes economical. That is, where $C^a < C^m$.

$$N > \frac{C_f^a}{C_v^m - C_v^a}.$$

This inequality provides insights into the importance of considering when to automate a process. IBM internal studies of software distribution have found that C_f^a can exceed 100 hours for complex packages. Our intuition based on a review of these data is that for complex installs, C_v^m is in the range of 10 to 20 hours, and C_v^a is in the range of 1 to 5 hours (mostly because of error recovery). Assuming that salaries are the same for all the staff involved, these numbers indicate that there should be approximately 5 to 20 targets for automated software distribution to be cost effective. In terms of the data in Figure 2, these numbers mean that from 15% to 30% of the installs should not have been automated.

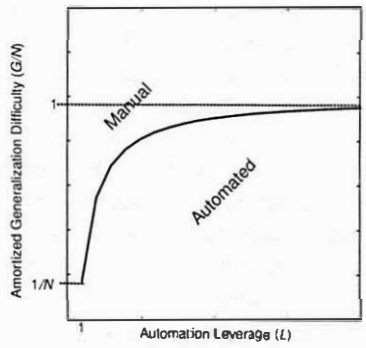


Figure 3: *Preference regions for automated and manual processes. Automated processes are preferred if there is a larger leverage for automation and/or if there is a smaller (amortized) difficulty of generalizing the manual procedure to an automated procedure (G/N).*

The foregoing cost models can be generalized further to obtain a broader understanding of the trade-off between manual and automated processes. In essence, this is a trade-off between the leverage provided by automation versus the difficulty of generalizing a manual process to an automated process.

Leverage L describes the factor by which the variable costs are reduced by using automation. That is, $L = \frac{C_v^m}{C_v^a} \geq 1$.

The generalization difficulty G relates to the challenges involved with designing, implementing, and testing automated versions of manual processes. Quantitatively, G is computed as the ratio between the fixed cost of automation and the variable cost of the manual process: $G = \frac{C_f^a}{C_v^m} \geq 1$. The intuition behind G is that, to construct an automated process, it is necessary to perform the manual process at least once. Any work beyond that test invocation of the manual process will result in a larger G. Substituting and solving, we find that

$$\frac{G}{N} = 1 - \frac{1}{L}$$

We refer to G/N as the amortized difficulty of generalization since the generalization difficulty is spread across N invocations of the automated process.

Figure 3 plots G/N versus L. We see that the vertical axis (G/N) ranges from $1/N$ to 1 since $G \geq 1$ and $G \leq N$. The latter constraint arises because there is little point in constructing automation that is G times more costly than a manual process if the process will only be invoked $N < G$ times. The figure identifies re-

gions in the $(L, G/N)$ space in which manual and automated processes are preferred. We see that if automation leverage is large, then an automated process is cost effective even if amortized generalization difficulty is close to 1. Conversely, if amortized generalization difficulty is small (close to $1/N$), then an automated process is cost effective even if automation leverage is only slightly more than 1. Last, having a longer process lifetime N means that G/N is smaller and hence makes an automated process more desirable.

This analysis suggests three approaches to reducing the cost of IT operations through automation: reduce the generalization difficulty G, increase the automation leverage L, and increase the process lifetime N. In the case of software distribution, the most effective approaches are to increase N and to reduce G. We can increase N by making the IT environment more uniform in terms of the types of hardware and software so that the same package can be distributed to more targets. However, two issues arise. First, increasing N has the risk of increasing the impact of automation failures, causing a commensurate decrease in L. Second, attempts to increase homogeneity may encounter resistance—ignoring a lesson learned from the transition from mainframes to client-server systems in the late 1980s, which was in large part driven by the desire of departments to have more control over their computing environments and hence a need for greater diversity.

To reduce cost by reducing G, one approach is to adopt the concept of mass customization developed in the manufacturing industry (e.g., [9]). This means designing components and processes so as to facilitate customization. In terms of software distribution, this might mean developing re-usable components for software packages. It also implies improving the reusability of process components—for example by standardizing the manual steps used in software package installations—so that a given automation technology can be directly applied to a broader set of situations. This concept of mass-customizable automated process components represents an important area of future research.

Mass customization can also be improved at the system level by having target systems that automatically discover their configuration parameters (e.g., from a registry at a well known address). This would mean that many differences between packages would be eliminated, reducing G and potentially leading to consolidation of package versions, also increasing N.

4 Related Work

The automation of IT operations has been a focus of attention for the last two decades [10], with on-going development of new technologies [5, 19, 2] and dozens of automation related products on the market [18]. More recently, there has been interest in process automation through workflow based solutions [6, 17, 14]. However, none of these efforts address the question of when automation reduces cost. There has been considerable interest in manufacturing in business cases for automation [12, 3, 7], and even an occasional study that addresses automation of IT operations [11, 15]. However, these efforts only consider the automation infrastructure, not whether a particular process with a limited lifetime should be automated.

5 Next Steps

One area of future work is to explore a broader range of IT processes so as to assess the generality of the automation analysis framework that we developed in the context of software distribution. Candidate processes to study include incident reporting and server configuration. The focus of these studies will be to assess (a) what automation is possible, (b) what additional processes are needed to support the automation, and (c) the fixed and variable costs associated with using automation on an on-going basis. Our current hypothesis for (b) is that additional processes are required for (1) preparing inputs, (2) invoking and monitoring the automation, (3) handling automation failures, and (4) maintaining the automation infrastructure. A particularly interesting direction will be to understand if there are any common patterns to the structure and cost of these additional processes across automation domains.

Thus far, we have discussed what automation should be done. Another consideration is the adoption of automation. Our belief is that SAs require a level of trust in the automation before the automation will be adopted. Just as with human relationships, trust is gained through a history of successful interactions. However, creating such a history is challenging because many of the technologies for IT automation are immature. As a result, care must be taken to provide incremental levels of automation that are relatively mature so that SA trust is obtained. One further consideration in gaining trust in automation is that automation cannot be a "black box" since gaining trust depends in part on SAs having a clear understanding of how the automation works.

The history of the automobile provides insight into the progression we expect for IT automation. In the early twentieth century, driving an automobile required considerable mechanical knowledge because of the need to make frequent repairs. However, today automobiles are sufficiently reliable so that most people only know that automobiles often need gasoline and occasionally need oil. For the automation of IT operation, we are at a stage similar to that of the early days of the automobile in that most computer users must also be system administrators (or have one close at hand). IT operations will have ma-

tured when operational details need not be surfaced to end users.

6 Conclusions

Recapping our position, we argue against the widely-held belief that automation always reduces the high costs of IT operations. Our argument rests on three pillars:

1. Introducing automation creates extra processes to deploy and maintain that automation, as we saw in comparing manual and automated software distribution processes.
2. Automation requires structured inputs (e.g., packages for a software distribution system) that introducing extra up-front (fixed) costs for design, implementation, and testing compared to manual processes. These fixed costs are a significant consideration in IT operations since many processes have a limited lifetime (e.g., a software package is installed on only a limited number of targets). Indeed, our studies of automated software distribution in several data centers found that 25% of the software packages were installed on fewer than 15 servers.
3. Detecting and removing errors from an automated process is considerably more complicated than for a manual process. Our software distribution data suggest that errors in automation can be frequent—19% of the requested installs failed in the data centers we studied.

Given these concerns, it becomes much less clear when automation should be applied. Indeed, in our model-driven analysis of software distribution in several large data centers, we found that 15–30% of automated software installs may have been less costly if performed Manually. Given that IT operations costs dominate IT spending today, it is essential that the kind of process-based analysis we have demonstrated here become an integral part of the decision process for investing in and deploying IT automation. We encourage the research community to focus effort on developing tools and more sophisticated techniques for performing such analyses.

References

[1] L. Bainbridge. The ironies of automation. In J. Rasmussen, K. Duncan, and J. Leplat, editors, *New Technology and Human Error*. Wiley, 1987.

[2] G. Candea, E. Kiciman, S. Kawamoto, and A. Fox. Autonomous recovery in componentized internet applications. *Cluster Computing Journal*, 2004.

[3] T.J. Caporello. Staying ahead in manufacturing and technology-the development of an automation cost of ownership model and examples. *IEEE International Symposium on Semiconductor Manufacturing*, 1999.

[4] D. Cappuccio, B. Keyworth, and W. Kirwin. Total Cost of Ownership: The Impact of System Management Tools. Technical report, The Gartner Group, 2002.

[5] G. Kaiser, J. Parekh, P. Gross, and G. Valetto. Kinesthetics extreme: An external infrastructure for monitoring distributed legacy systems. In *Fifth Annual International Active Middleware Workshop*, 2003.

[6] A. Keller, J.L. Hellerstein, J.L. Wolf, K.-L. Wu, and V. Krishnan. The champs system: Change management with planning and scheduling. In *IEEE/IFIP Network Operations and Management*, April 2004.

[7] N.S. Markushevich, I.C. Herejk, and R.E. Nielsen. Function requirements and cost-benefit study for distribution automation at B.C. Hydro. *IEEE International Transactions on Power Systems*, 9(2):772–781, 1994.

[8] MicroData. The Hidden Cost of Your Network. Technical report, MicroData, 2002.

[9] J.H. Mikkola and T. Skjott-Larsen. Supply-chain integration: implications for mass customization, modularizaiton and postponement strategies. *Production Planning and Control*, 15(4):352–361, 2004.

[10] K.R. Milliken, A.V. Cruise, R.L. Ennis, A.J. Finkel, J.L. Hellerstein, D.J. Loeb, D.A. Klein, M.J. Masullo, H.M. Van Woerkom, and N.B. Waite. YES/MVS and the autonomation of operations for large computer complexes. *IBM Systems Journal*, 25(2), 1986.

[11] NetOpia. netoctopus: The comprehensive system administration solution. `http://www.netopia.com/software/pdf/net0-ROI.pdf`, 2005.

[12] C.A. Niznik. Cost-benefit analysis for local integrated facsimile/data/voice packet communication networks. *IEEE Transactions on Communications*, 30(1), January 1982.

[13] UK Office of Government Commerce. *Best Practice for Service Support*. IT Infrastructure Library Series. Stationery Office, 1st edition, 2000.

[14] Peregrine. Service center. `http://www.peregrine.com/products/servicecenter.asp`, 2005.

[15] M. H. Sherwood-Smith. Can the benefits of integrated information systems (IIS) be costed. *Internation Conference on Information Technology in the Workplace*, pages 11–18, 1991.

[16] Serenity Systems. Managed Clinent Impact on the Cost of Computing. `http://www.serenity-systems.com`, 2005.

[17] G. Valetto and G. Kaiser. A case study in software adaptation. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 73–78, 2002.

[18] ComputerWorld Staff Writer. E-business buyers' guide. In `www.computerworld.com`, 2005.

[19] S. A. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie. High speed and robust event correlation. *IEEE Communications Magazine*, 34(5):82–90, 1996.

Human-Aware Computer System Design *

Ricardo Bianchini, Richard P. Martin, Kiran Nagaraja, Thu D. Nguyen, Fábio Oliveira
Department of Computer Science, Rutgers University, Piscataway, NJ 08854
{*ricardob, rmartin, knagaraj, tdnguyen, fabiool*}*@cs.rutgers.edu*

Abstract

In this paper, we argue that human-factors studies are critical in building a wide range of dependable systems. In particular, only with a deep understanding of the causes, types, and likelihoods of human mistakes can we build systems that prevent, hide, or at least tolerate human mistakes by design. We propose several research directions for studying how humans impact availability in the context of Internet services. In addition, we describe validation as one strategy for hiding human mistakes in these systems. Finally, we propose the use of operator, performance, and availability models to guide human actions. We conclude with a call for the systems community to make the human an integral, first-class concern in computer system design.

1 Introduction

As computers permeate all aspects of our lives, a wide range of computer systems must achieve high dependability, including availability, reliability, and security. Unfortunately, few current computer systems can legitimately claim to be highly dependable. Further, many studies over the years have empirically observed that human mistakes are a large source of unavailability in complex systems [7, 13, 15, 16]. We suspect that many security vulnerabilities are also the result of mistakes, but are only aware of one study that touches on this issue [15].

To address human mistakes and reduce operational costs, researchers have recently started to design and implement autonomic systems [9]. Regardless of how successful the autonomic computing effort eventually becomes, humans will always be part of the installation and management of complex computer systems at some level. For example, humans will likely always be responsible for determining a system's overall policies, for addressing any unexpected behaviors or failures, and for upgrading software and hardware. Thus, human mistakes are inevitable.

In this paper, we argue that human mistakes are so common and harmful because computer system designers have consistently failed to consider the human-system interaction explicitly. There are at least two reasons for this state of affairs. First, dependability is often given a lower priority than other concerns, such as time-to-market, system features, performance, and/or cost, during the design and implementation phases. As a result, improvements in dependability come only after observing failures of deployed systems. Indeed, one need not look past the typical desktop to see the results of this approach. Second, understanding human-system interactions is time-consuming and unfamiliar, in that it requires collecting and analyzing behavior data from extensive human-factors experiments.

Given these observations, we further argue that dependability and, in particular, the effect of humans on dependability should become a first-class design concern in complex computer systems. More specifically, we believe that human-factors studies are necessary to identify and understand the causes, types, and likelihoods of human mistakes. By understanding human-system interactions, designers will then be able to build systems to avoid, hide, or tolerate these mistakes, resulting in significant advances in dependability.

In the remainder of the paper, we first briefly consider how designers of safety-critical systems have dealt with the human factor in achieving high dependability. We also touch on some related work. After that, we propose several research directions for studying how human mistakes impact availability in the context of Internet services. We then describe how *validation* can be used to hide mistakes and *guidance* to prevent or at least mitigate the impact of mistakes. Finally, we speculate on how a greater understanding of human mistakes can improve the dependability of other areas of computer systems.

*This research was partially supported by NSF grants #EIA-0103722, #EIA-9986046, and #CCR-0100798.

2 Background and Related Work

Given the prominent role of human mistakes in system failures, human-factors studies have long been an important ingredient of engineering safety-critical systems such as air traffic and flight control systems, e.g., [6, 18]. In these domains, the enormous cost of failures requires a significant commitment of resources to accounting for the human factor. For example, researchers have often sought to understand the mental states of human operators in detail and create extensive models to predict their actions. Our view is that system designers must account for the human factor to achieve high dependability but at lower costs than for safety-critical systems. We believe that this goal is achievable by focusing on human mistakes and their impact on system dependability, rather than attempting a broader understanding of human cognitive functions.

Our work is complementary to research on Human-Computer Interaction (HCI), which has traditionally focused on ease-of-use and cognitive models [17], in that we seek to provide infrastructural support to ease the task of operating highly available systems. For example, Barrett *et al.* report that one reason why operators favor using command line interfaces over graphical user interfaces is that the latter tools are often less trustworthy (e.g., their depiction of system state is less accurate) [1]. This suggests that HCI tools will only be effective when built around appropriate infrastructure support. Our vision of a runtime performance model that can be used to predict the impact of operator actions (Section 5) is an example of such infrastructural support. Further, our validation infrastructure will provide a "safety net" that can hide human mistakes caused by inexperience, stress, carelessness, or fatigue, which can occur even when the HCI tools provide accurate information.

Curiously, we envision guidance techniques that may even appear to conflict with the goals of HCI at first glance. For example, we plan to purposely add "inertia" to certain operations to reduce the possibility of serious mistakes, making it more difficult or time-consuming to perform these operations. Ultimately however, our techniques will protect systems against human mistakes and so they are compatible with the HCI goals.

Our work is related to several recent studies that have gathered empirical data on operator behaviors, mistakes, and their impact on systems [1, 16]. Brown and Patterson have proposed a methodology to consider humans in dependability benchmarking [4] and studied the impact of *undo*, an approach that is orthogonal (and complementary) to our validation and guidance approach, on repair times for several faults injected into an email service [3]. To our knowledge, however, we were the first group to publish detailed data on operator mistakes [15].

Our work is also related to no-futz computing [8]. However, we focus on increasing availability, whereas no-futz computing seeks to reduce futzing and costs.

3 Operator Mistakes

In order to build systems that reduce the possibility for operator mistakes, hide the mistakes, or tolerate them, we must first better understand the nature of mistakes. Thus, we believe that the systems community must develop common benchmarks and tools for studying human mistakes [4]. These benchmarks and tools should include infrastructure for experiment repeatability, e.g. instrumentation to record human action logs that can later be replayed. Finally, we need to build a shared body of knowledge on what kind of mistakes occur in practice, what their causes are, and how they impact performance and availability.

We have already begun to explore the nature of operator mistakes in the context of a multi-tier Internet service. In brief, we asked 21 volunteer operators to perform 43 benchmark operational tasks on a three-tier auction service. Each of the experiments involved either a scheduled-maintenance task (e.g., upgrading a software component) or a diagnose-and-repair task (e.g., discovering a disk failure and replacing the disk). To observe operator actions, we asked the operators to use a shell that records and timestamps every command typed into it and the corresponding result. Our service also recorded its throughput throughout each experiment so that we could later correlate mistakes with their impact on service performance and availability. Finally, one of our team members personally monitored each experiment and took notes to ease the interpretation of the logged commands and to record observables not logged by our infrastructure, such as edits of configuration files.

We observed a total of 42 mistakes, ranging from software misconfiguration, to fault misdiagnosis, to software restart mistakes. We also observed that a large number of mistakes (19) led to a degradation in service throughput. These results can now be used to design services that can tolerate or hide the mistakes we observed. For example, we were able to evaluate a prototype of our validation approach, which we describe in the next section.

We learned several important lessons from this experience: First, although we scripted much of the setup for each experiment, most of the scripts were not fully automated. This was a mistake. On several occasions, we only caught mistakes in the manual part of the setup just before the experiment began. Finding human subjects is too costly to risk invalidating any experiment in this manner. Second, infrastructural support for viewing the changes made to configuration files would have been very helpful. Third, we used a single observer for

all of our experiments, which in retrospect, was a good decision because it kept the human recorded data as consistent as possible across the experiments. However, on several occasions, our observer scheduled too many experiments back-to-back, making fatigue a factor in the accuracy of the recorded observations. Fourth, our study was time-consuming. Even seemingly simple tasks may take operators a long time to complete; our experiments took an average of 1 hour and 45 minutes each. We also ran 6 warm up experiments to allow some of the novice operators to become more familiar with our system; these took on average 45 minutes each. Combining the different sources of data and analyzing them were also effort-intensive. Finally, enlisting volunteer operators was not an easy task. Indeed, one of the shortcomings of our study is the dearth of experienced operators among our volunteer subjects.

Despite these difficulties, our study (along with [1, 3]) proves that performing human-factor studies is not intractable for systems researchers. In fact, these studies should become easier to perform over time, as researchers share their tools, data, and experience with human-factor studies.

3.1 Open Issues

While our initial study represents a significant first step, it also raises many open issues.

Effects of long-term interactions. The short duration of our experiments meant that we did not account for a host of effects that are difficult to observe at short timescales. For example, the effect of increasing familiarity with the system, the impact of user expectations, systolic load variations, stress and fatigue, and the impact of system evolution as features are added and removed.

Impact of experience. 14 of our 21 volunteer operators were graduate students with limited experience with the operation of computing services; 11 of the 14 were classified as novices, while 3 were classified as intermediates (on a three-tier scale: novice, intermediate, expert).

Impact of tools and monitoring infrastructures. Our study did not include any sophisticated tools to help with the service operation; we only provided our volunteers with a throughput visualization tool. Operators of real services have a wider set of monitoring and maintenance tools at their disposal.

Impact of complex tasks. Our experiments covered a small range of fairly simple operator tasks. Difficult tasks such as dealing with multiple overlapping component faults and changing the database schema that intuitively might be sources of more serious mistakes have not been studied.

Impact of stress. Many mistakes happen when humans are operating under stress, such as when trying to repair parts of a site that are down or under attack. Our initial experiments did not consider these high-stress situations.

Impact of realistic workloads. Finally, the workload offered to the service in our experiments was generated by a client emulator. It is unclear whether the emulator actually behaves as human users would and whether client behavior has any effect on operator behavior.

3.2 Current and Future Work

Encouraged by our positive initial experience, we are currently planning a much more thorough study of operator actions and mistakes. In particular, we plan to explore three complimentary directions: (1) survey and interview experienced operators, (2) improve our benchmarks and run more experiments, and (3) run and monitor all aspects of a real, live service for at least one year. The surveys and interviews will unearth the problems that afflict experienced operators even in the presence of production software and hardware and sophisticated support tools. This will enable us to design better benchmarks as well as guide our benchmarking effort to address areas of maximum impact. Running a live service will allow us to train the operators extensively, observe the effects of experience, stress, complex tasks, and real workloads, and study the efficacy of software designed to prevent, hide, or tolerate mistakes.

We have started this research by surveying professional network and database administrators to characterize the typical administration tasks, testing environments, and mistakes. Thus far, we have received 41 responses from network administrators and 51 responses from database administrators (DBAs). Many of the respondents seemed excited by our research and provided extensive answers to our questions. Thus, we believe that the challenge of recruiting experienced operators for human-factor studies is surmountable with an appropriate mix of financial rewards and positive research results.

A synopsis of the DBAs' responses follows. All respondents have at least 2 years of experience, with 71% of them having at least 5 years of experience. The most common tasks, accounting for 50% of the tasks performed by DBAs, relate to recovery, performance tuning, and database restructuring. Interestingly, only 16% of the DBAs test their actions on an exact replica of the online system. Testing is performed offline, manually or via ad-hoc scripts, by 55% of the DBAs. Finally, DBA mistakes are responsible (entirely or in part) for roughly 80% of the database administration problems reported. The most common mistakes are deployment, performance, and structure mistakes, all of which occur once per month on average. The current differences

and separation between offline testing and online environments are cited as two of the main causes of the most frequent mistakes. These results further motivate the validation and guidance approaches discussed next.

4 Validation

In this section, we describe validation as one approach for hiding mistakes. Specifically, we are prototyping a *validation* environment that allows operators to validate the correctness of their actions before exposing them to clients [15]. Briefly, our validation approach works as follows. First, each component that will be affected by an operator action is taken offline, one at a time. All requests that would be sent to the component are redirected to components that provide the same functionality but that are unaffected by the operator action. After the operator action has been performed, the affected component is brought back online but is placed in a sand-box and connected to a validation harness. The validation harness consists of a library of real and proxy components that can be used to form a virtual service around the component under validation. The harness requires only a few machines and, thus, has negligible resource requirements for real services. Together, the sand-box and validation harness prevent the component, called *masked* component, from affecting the processing of client requests while providing an environment that looks exactly like the live environment.

The system then uses the validation harness to compare the behavior of the component affected by the operator action against that of a similar but unaffected component. If this comparison fails, the system alerts the operator before the masked component is placed in active service. The comparison can either be against another live component, or against a previously collected trace. After the component passes the validation process, it is migrated from the sand-box into the live operating environment without any changes to its configurations.

Using our prototype validation infrastructure, we were able to detect and hide 66% of the mistakes we observed in our initial human-factors experiments. A detailed evaluation of our prototype can be found in [15].

4.1 Open Issues

Although our validation prototype represents a good first step, we now discuss several open issues.

Isolation. A critical challenge is how to isolate the components from each other yet allow them to be migrated between live and validation environments with no changes to their internal state or to external configuration parameters, such as network addresses. We can achieve this isolation and transparent migration at the granularity of an entire node by running nodes over a virtual network, yet for other components this remains a concern.

State management. Any validation framework is faced with two state management issues: (1) how to start up a masked component with the appropriate internal state; and (2) how to migrate a validated component to the online system without migrating state that was built up during validation but is not valid for the live service.

Bootstrapping. A difficult open problem for validation is how to check the correctness of a masked component when there is no component or trace to compare against. This problem occurs when the operator action correctly changes the behavior of the component for the first time.

Non-determinism. Validation depends on good comparator functions. Exact-match comparator functions are simple but limiting because of application nondeterminism. For example, ads that should be placed in a Web page may correctly change over time. Thus, some relaxation in the definition of similarity is often needed, yet such relaxation is application-specific.

Resource management. Regardless of the validation technique and comparator functions, validation retains resources that could be used more productively when no mistakes are made. Under high load, when all available resources should be used to provide a better quality of service, validation attempts to prevent operator-induced service unavailability at the cost of performance. This suggests that adjusting the length of the validation period according to load may strike an appropriate compromise between availability and performance.

Comprehensive validation. Validation will be most effective if it can be applied to all system components. To date, our prototyping has been limited to the validation of Web and application servers in a three-tier service. Designing a framework that can successfully validate other components, such as databases, load balancers, switches, and firewalls, presents many more challenges.

4.2 Current Work

We are extending our validation framework in two ways to address some of the above issues. First, we are extending our validation techniques to include the database, an important component of multi-tier Internet services. Specifically, we are modifying a replicated database framework, called C-JDBC, which allows for mirroring a database across multiple machines. We are facing several challenges, such as the management of the large persistent state when bringing a masked database up-to-date, and the performance consequences of this operation.

Second, we are considering how to apply validation

when we do not have a known correct instance for comparison. Specifically, we are exploring an approach we call *model-based* validation. The idea is to validate the system behavior resulting from an operator action against an operational model devised by the system designer. For example, when configuring a load balancing device, the operator is typically attempting to even out the utilization of components downstream from the load balancer. Thus, if we can conveniently express this resulting behavior (or model) and check it during validation, we can validate the operator's changes to the device configuration. We are currently designing a language that can express such models for a set of components, including load balancers, routers, and firewalls.

5 Guidance

In this section, we consider how services can prevent mistakes by *guiding* operator actions when validation is not applicable. For example, when the operator is trying to restore service during a service disruption, he may not have the leisure of validating his actions since repairs need to be completed as quickly as possible. Guidance can also reduce repair time by helping the operator to more rapidly find and choose the correct actions.

One possible strategy is to use the data gathered in operator studies to create models of operator behaviors and likely mistakes, and then build services that use these models together with models of the services' own behaviors to guide operator actions. In particular, we envision services that monitor and predict the potential impact of operator actions, provide feedback to the operator before the actions are actually performed, suggest actions that can reduce the chances for mistakes, and even require appropriate authority, such as approval from a senior operator, before allowing actions that might negatively impact the service.

5.1 Future Work

Our guidance strategy relies on the system to maintain several representations of itself: an operator model, a performance model, and an availability model.

Operator behavior models. To date, operator modeling has mostly been addressed in the context of safety-critical systems or those where the cost of human mistakes can be very high. Rather than follow the more complex cognitive approaches that have evolved in these areas (see Section 2), we envision a simpler approach in which the operator is modeled using stochastic state machines describing expected operator behavior.

Our intended approach is similar in spirit to the Operation Function Models (OFMs) first proposed in [12]. Like the OFMs, our models will be based on finite automata with probabilistic transitions of operator actions, which can be composed hierarchically. However, we do not plan on representing the mental states of the operator, nor do we expect to model the operator under normal operating conditions.

An important open issue to be considered is whether tasks are repeated enough times with sufficient similarity to support the construction of meaningful models. In the absence of a meaningful operator model for a certain task, we need to rely on the other models for guidance.

Predicting the impact of operator actions. Along with the operator behavior models, we will need a software monitoring infrastructure for the service to represent itself. In particular, it is important for the service to monitor the configuration and utilization of its hardware components. This information can be combined with analytical models of performance and availability similar to those proposed in [5, 14] to predict the impact of operator actions. For example, the performance (availability) model could estimate the performance (availability) degradation that would result from taking a Web server into the validation slice for a software upgrade.

Guiding and constraining operator actions. Using our operator models, we will develop software to guide operator actions. Guiding the operator entails assisting him/her in selecting actions likely to address a specific scenario. These correspond to what today might be entries in an operations manual. However, unlike a manual, our guidance system can directly observe current system state and past action history in suggesting actions.

Our approach to guide the operator uses the behavior models, the monitoring infrastructure, and the analytical models to determine the system impact of each action. Given a set of behavior model transitions, the system can suggest the operator actions that are least likely to cause a service disruption or performance degradation. To do so, the system will first determine the set of components that are likely to be affected by each operator action and the probability that these components would fail as a result of the action. The system will then predict the overall impact for each possible action along with the likelihoods of each of these scenarios.

To allow operators to deviate from automatic guidance yet allow a service to still protect itself against arbitrary behaviors, we will need *dampers*. The basic idea behind the damper is to introduce inertia representing the potential negative impact of an operator's action in case the action is a mistake. For example, if an action is likely to have a small negative (performance or availability) impact on the service, the damper might simply ask the operator to verify that he indeed really wants to perform that action. On the other hand, if the potential impact

of the operator's action is great enough, the system may require the intervention of a senior or "master" operator before allowing the action to take place. In a similar vein, Bhaskaran *et al.* [2] have recently argued that systems should require acknowledgements from operators before certain actions are performed. However, the need for acknowledgements in their proposed systems would be determined by operator behavior models only.

6 Discussion and Conclusion

The research we have advocated in this paper is applicable to many other areas of Computer Science. In this section, we motivate how some of these areas may be improved by accounting for human actions and mistakes.

In the area of Operating Systems, little or no attention has been paid to how mistakes can impact the system. For example, when adding a device driver, a simple mistake can bring down the system. Also, little attention has been given to the mistakes made when adding and removing application software. Addressing these mistakes explicitly would increase robustness and dependability.

In the area of Software Engineering, again historically there has been little direct investigation into why and how people make mistakes. A small body of work exists in examining common types of programming errors, yet little is understood about the processes that cause there errors. An interesting example of work in this direction is [10], in which the authors exploit data mining techniques to detect cut-and-paste mistakes.

Finally, in the field of Computer Networks, the Border Gateway Routing Protocol suffered from severe disruptions when bad routing entries were introduced, mostly as a result of human mistakes [11]. Again, addressing human mistakes explicitly in this context can significantly increase routing robustness and dependability.

In conclusion, we hope that this paper included enough motivation, preliminary results, and research directions to convince our colleagues that designers must consider human-system interactions and the mistakes that may result explicitly in their designs. In this context, human-factors studies, techniques to prevent or hide human mistakes, and models to guide operator actions all seem required. Failure to address humans explicitly will perpetuate the current scenario of human-produced unavailability and its costly and annoying consequences.

References

[1] BARRETT, R., MAGLIO, P. P., KANDOGAN, E., AND BAILEY, J. Usable Autonomic Computing Systems: the Administrator's Perspective. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC'04)* (May 2004).

[2] BHASKARAN, S. M., IZADI, B., AND SPAINHOWER, L. Coordinating Human Operators and Computer Agents for Recovery-Oriented Computing. In *Proceedings of the International Conference on Information Reuse and Integration* (Nov. 2004).

[3] BROWN, A. *A Recovery-oriented Approach to Dependable Services: Repairing Past Errors with System-wide Undo.* PhD thesis, Computer Science Division-University of California, Berkeley, 2003.

[4] BROWN, A., AND PATTERSON, D. A. Including the Human Factor in Dependability Benchmarks. In *Proceedings of the DSN Workshop on Dependability Benchmarking* (June 2002).

[5] CARRERA, E. V., AND BIANCHINI, R. Efficiency vs. Portability in Cluster-Based Network Servers. In *Proceedings of the 8th Symposium on Principles and Practice of Parallel Programming (PPoPP)* (June 2001).

[6] GOVINDARAJ, T., WARD, S. L., POTURALSKI, R. J., AND VIKMANIS, M. M. An Experiment and a Model for the Human Operator in a Time-Constrained Competing-Task Environment. *IEEE Transactions on Systems Man and Cybernetics 15*, 4 (1985).

[7] GRAY, J. Why do Computers Stop and What Can Be Done About It? In *Proceedings of 5th Symposium on Reliability in Distributed Software and Database Systems* (Jan. 1986).

[8] HOLLAND, D. A., JOSEPHSON, W., MAGOUTIS, K., SELTZER, M. I., STEIN, C. A., AND LIM, A. Research Issues in No-Futz Computing. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)* (May 2003).

[9] KEPHART, J. O., AND CHESS, D. M. The Vision of Autonomic Computing. *IEEE Computer 36*, 1 (Jan. 2003).

[10] LI, Z., LU, S., MYAGMAR, S., AND ZHOU, Y. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)* (Dec. 2004).

[11] MAHAJAN, R., WETHERALL, D., AND ANDERSON, T. Understanding BGP Misconfiguration. In *Proceedings of the ACM SIGCOMM '02 Conference on Communications Architectures and Protocols* (Aug. 2002).

[12] MITCHELL, C. M. GT-MSOCC: A Domain for Research on Human-Computer Interaction and Decision Aiding in Supervisory Control Systems. *IEEE Transactions on Systems, Man and Cybernetics 17*, 4 (1987), 553–572.

[13] MURPHY, B., AND LEVIDOW, B. Windows 2000 Dependability. Tech. Rep. MSR-TR-2000-56, Microsoft Research, June 2000.

[14] NAGARAJA, K., GAMA, G., MARTIN, R. P., JR., W. M., AND NGUYEN, T. D. Quantifying Performability in Cluster-Based Services. *IEEE Transactions on Parallel and Distributed Systems 16*, 5 (May 2005).

[15] NAGARAJA, K., OLIVEIRA, F., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. D. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)* (Dec. 2004).

[16] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. Why do Internet Services Fail, and What Can Be Done About It. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS'03)* (Mar. 2003).

[17] RASMUSSEN, J. *Information Processing and Human-Machine Interaction: An Approach to Cognitive Engineering.* North-Holland, New York, 1986.

[18] WALDEN, R. S., AND ROUSE, W. B. A Queueing Model of Pilot Decisionmaking in a Multitask Flight Management Situation. *IEEE Transactions on Systems, Man and Cybernetics 8*, 12 (Dec. 1978).

Thirty Years is Long Enough: Getting Beyond C

Eric Brewer Jeremy Condit Bill McCloskey Feng Zhou

Computer Science Division, University of California at Berkeley
{brewer,jcondit,billm,zf}@cs.berkeley.edu

Abstract

Thirty years after its creation, C remains one of the most widely used systems programming languages. Unfortunately, the power of C has become a liability for large systems projects, which are now focusing on security and reliability. Modern languages and static analyses provide an opportunity to improve the quality of systems software, and yet adoption of these tools has been slow.

To address this problem, we propose a new language called Ivy that has an evolutionary path from C. The mechanism for this evolutionary path is a system of *extensions* and *refactorings*: extensions augment the language with new features, and refactorings assist the programmer in updating their code to use these new features. Extensions and refactorings have a wide variety of applications, from enforcing memory safety to detecting user/kernel pointer errors. We also demonstrate Macroscope, a tool we have built to enable refactoring of existing C code.

1 Introduction

Since the time of their creation, the relationship between Unix and C has been symbiotic: C matured because of its link to Unix, and Unix flourished because C was a quantum leap beyond its predecessor, assembly language. Thirty years after its creation, C is now deeply entrenched in the operating system community—but it is showing its age. We believe that good languages lead to good systems; thus, it is time for new language technology to drive new systems research. Unfortunately, rescuing existing systems from the perils of C is non trivial.

One possible approach to improving language technology for systems is to focus on an entirely new language. Modern languages such as Java and ML provide stronger static guarantees, such as type and memory safety, at a slight cost in expressiveness. This trade-off may be desirable for some systems, which emphasize reliability, security, and availability over raw performance. However, these languages lack a number of useful features of C, such as manual memory management and bit-level data layout. Also, it is impractical to rewrite existing systems in an entirely new language—with millions of lines of C code running critical infrastructure, we cannot afford to simply start over.

A second possible approach to this problem is to use static analysis to root out software problems. The benefit of analysis is that it finds bugs without requiring code to be rewritten in a new language or a new model. However, static analysis tools are difficult to write and often difficult to use. Since C imposes no restrictions on where and when programs can write to memory, tools must make very conservative assumptions about program behavior, or else pay a huge cost in the complexity of the analysis. And because all analyses are conservative in some way, they usually yield large numbers of false positives, which make real bugs more difficult to detect. These false positives, combined with long analysis times, make it difficult to integrate static analysis directly into the build process of a program, which in turn hinders the ability of these tools to have a lasting impact on source code quality.

We propose a third approach that offers an *evolutionary path* from C to a new language called Ivy. This approach incorporates the advantages of both of the previous ones. First, Ivy is a programming language as opposed to an analysis tool; it will provide sound guarantees to programmers using a checker that will be integrated into the compiler. Second, Ivy will provide a transition path from existing code by means of *extensions* and *refactorings*. Extensions will add new language features such as sophisticated data layout, concurrency control, and memory management, each of which can be enabled or disabled individually. Extensions may add language features, but they may also disable them. For example, the memory safety extension will forbid some uses of casting and pointer arithmetic while adding mechanisms such as regions and built-in reference counting. Refactorings[1] will assist programmers in the transi-

[1] The traditional definition of "refactoring" implies a structural improvement that preserves semantic meaning; we use the term more broadly in that we allow small changes in semantics, such as the addition of type safety.

tion by analyzing existing code to find patterns that could be better expressed with a specific language extension. Working in tandem, extensions and refactorings will enable a transition to modern language features; indeed, they will also allow future evolution as new language technology emerges.

This paper presents our vision for the future of systems programming. First, we discuss the problems of C, and we describe a number of features that we would like to see in a new systems programming language (Section 2). Then, we discuss in more detail our evolutionary path toward this new language (Section 3) as well as a few examples of extensions and refactorings (Section 4). Finally, we present our initial results, which demonstrate that it is possible to migrate code to a more solid foundation and to apply useful refactorings (Section 5) with modest programmer effort.

2 Requirements for a Replacement

C succeeded for many years because systems written in C were safer, more portable, and more maintainable than those written in assembly. Equally important, C programs performed nearly as well as their assembly counterparts. But as time has passed, new languages have picked up the standards of safety and reliability, while C has not progressed. The most obvious gap has been in memory safety, where languages like Java and ML provide much stronger guarantees than C. Less obviously, but just as important, C fails to provide the programmer with tools for concurrency control, safe data layout, and other system-specific tasks.

In this section, we discuss some of the key features that we would like to see in a successor to C. We believe that these changes will have a positive impact on the safety and security of systems programs.

Type and memory safety. Memory safety is a crucial property for safe and secure systems. The Cyclone language [10] permits programs to use a number of safe, flexible memory management policies, such as region-based memory management, reference counting, and garbage collection, all within the context of a C-like language. Also, the CCured tool [3] analyzes pointer usage to introduce efficient run-time memory safety checks. These tools demonstrate that memory safety is a reasonable goal in a C-like language.

Besides catching bugs, type safety makes other analyses easier to write. In a type safe language, two memory locations cannot be aliased if their types differ. C lacks this property, making it more difficult to develop tools. Memory management disciplines like regions also make analysis easier, since they refine the type system further, reducing possible aliasing relationships. In general, we believe that increased memory safety will have the additional benefit of making programs easier to analyze.

Concurrency. A modern systems programming language must have native support for concurrency for a number of reasons. First of all, integrating threads and atomic sections [6] into the language makes it easier for programmers to write safe and portable code. Indeed, Boehm has shown that implementing threads without some compiler support is unsafe [2]. Secondly, built-in support for concurrency makes it easier for the compiler to check properties of concurrent programs. Most tools have difficulty processing concurrent software, since they fail to take into account all possible thread interleavings; however, code written using atomic sections should be easier to analyze, since the interactions between threads are spelled out explicitly. The Calvin-R checker makes use of atomicity in this way [8].

API adherence. Systems software often must comply with complex interfaces. Tools such as the metacompilation (MC) system [9], SLAM [1], and ESP [4] ensure that code adheres to a given interface. Unfortunately, these tools have difficulty with pointers and aliasing. We believe that some of these problems can be eliminated with the introduction of stronger type systems and memory management disciplines in the language, as mentioned above.

Data layout. Modern languages that strictly enforce type safety, such as Java and ML, almost always require data to be formatted according to conventions specified by the language or the compiler. C allows *a priori* data layout, where the programmer can control data formatting down to the bit level, which is important for systems applications that must be compatible with existing libraries, file formats, or network protocols. Unfortunately, data layout in C is not safe, so we intend to supply a mechanism to allow type-safe *a priori* data layout using a dependent type system [14]. Dependent type systems have been studied in the context of functional languages, but new research is necessary to make them work in an imperative language like C.

3 The Ivy Platform

In general, new technologies often fail for two very important reasons. The primary cause is the lack of an evolutionary path to the new platform. Although it is tempting to make a clean break with the past, users will rarely choose to adopt a technology that makes their old software obsolete. A secondary cause of failure is a lack of extensibility, which allows a platform to be updated to meet changing requirements. Therefore, the Ivy platform that we propose is both extensible and evolutionary.

New Ivy features will be implemented via language

extensions, which will plug into the compiler and provide new syntax, type checking rules, and code generation options. Language extensions will implement system-specific checking, similar to the checks provided by MC [9] or SLAM [1]. These extensions will also give the programmer flexibility in choosing what rules the compiler should check, since extensions may be enabled or disabled selectively. For example, programmers will not need to enable the memory safety extension until their code conforms to the specific rules about regions and reference counting required by that extension.

The Ivy platform will also include refactoring tools so that programmers can evolve legacy code. The first step in refactoring a program is to convert it to Ivy without any extensions enabled. We describe this step in detail in Section 5; in brief, the goal is to eliminate use of the C preprocessor, which is not present in Ivy. Once a C program has been converted to Ivy, refactoring tools will enable the use of new language extensions for code that was not originally written with those extensions in mind. For example, a tool might use a region inference algorithm to make region-based memory management explicit, thereby allowing the memory safety extension to certify the code as memory safe. Unlike language extensions, which are run each time the compiler is used, refactorings are applied only once in the lifetime of the code. We expect that each language extension will be bundled with a refactoring to enable that extension on legacy code. Refactorings may require a small amount of user guidance, but they will be mostly automatic.

This approach has several advantages over designing a new language or creating more bug-finding tools:

- There is no need for manual translation of programs. Languages like Java and ML provide attractive features like memory safety, but they require that existing C code be rewritten. Even Cyclone, which is similar to C, requires extensive manual intervention. Ivy's refactoring tools will make use of program analyses, like the one found in CCured [3], to make these changes automatically. User guidance will be necessary but relatively rare. Although refactorings may be somewhat heavy-weight, they will be only applied once. Our goal is to shift much of the burden of static analysis out of a compiler or checker, which is run frequently, to a single-use refactoring. Since they are only run once, refactorings will have a somewhat larger "budget" of running time and user interaction that traditional bug-finding tools.

- Language extensions will build on each other. We expect that the guarantees provided by one extension will be used by other extensions. For example, many program analyses for C assume memory safety in order to operate correctly. Ivy extensions can make this requirement explicit by depending on the Ivy memory safety extension. As a more concrete example, an extension for checking protocol adherence (like MC, SLAM, or ESP) would be much easier to write if it could assume that all memory accesses respect types, that data is segregated into explicit regions, and that concurrent memory accesses occur only inside atomic sections. These guarantees would be provided by underlying Ivy extensions for memory safety and concurrency.

- Programmers can choose which extensions they need. Refactorings might be difficult to apply, since they make substantial changes to source code. In some cases, it may not be possible to apply all extensions to an extremely old and baroque code base, even with the aid of refactorings. But since Ivy extensions will be enabled selectively, programmers may use only those that are practical.

- Researchers can develop custom extensions and refactorings. Since systems software is constantly changing, it may be necessary in the future to create new Ivy extensions, as well as the refactorings that enable them. Ivy will be built to make this process as simple as possible.

4 Extension Examples

In this section, we present two examples that show how our new programming platform will help developers to write better systems software. In particular, these examples demonstrate the power of automated refactorings and language extensions. These examples are only theoretical, but we believe that they represent realistic uses of our proposed language.

CCured. The CCured project [3, 11] analyzes pointer usage in large software systems in order to add low-cost memory safety checks. CCured infers a pointer "kind" for each pointer in the program, and this pointer kind is the basis for the static checks and run-time instrumentation performed by CCured. Unfortunately, a significant amount of manual intervention is required in order to "cure" real programs, since CCured sometimes fails to understand why a particular piece of code is memory-safe. Worse, this manual intervention must be repeated with each subsequent release of the software package that is being cured. With our framework, CCured will be implemented as an extension and a refactoring: the refactoring will infer pointer kinds to the best of its ability, and the extension component will do the corresponding type-checking and instrumentation every time the software

is compiled. Because the refactoring produces human-readable code, the programmer will have a chance to improve the results of CCured's analysis. Also, because this annotated code becomes the master copy in the repository, the programmer need not repeat this procedure after every update to the project.

Linux's Sparse. Linux creator Linus Torvalds wrote a static analysis tool called Sparse [12], which is specifically tailored for checking the Linux kernel. It uses extra annotations added by programmers in order to verify certain kernel-specific properties. For example, programmers can annotate pointers that should contain user-space addresses, and Sparse will verify that none of these pointers are dereferenced directly by the kernel. With our framework, a refactoring will discover pointers that contain user-space addresses (much like the existing CQual project [7]), and an extension will be responsible for checking, at each compilation, that these pointers are never dereferenced. This approach makes annotations explicit in the code and thus integrates this check into the development process.

5 Macroscope: A First Step

Although "vanilla" Ivy code, without any extensions, is similar to C, there are some differences. The most important one is the lack of the C preprocessor (CPP) in Ivy. CPP poses a great challenge for refactoring tools: because the preprocessor is token-based rather than syntax-tree-based, refactoring tools cannot parse CPP code directly. Since refactorings are so critical to the Ivy platform, the first step in the translation to Ivy is the elimination of CPP. In its place, Ivy includes a flexible macro system that is based on syntax trees rather than tokens. Thus, refactoring tools can operate on Ivy macros directly, without first expanding them. This feature is critical to the success of refactoring tools, since they must preserve the readability and human understanding of the code in order to be useful. To solve this problem, we have developed a tool called Macroscope, which transforms a C program into an Ivy program while preserving the vast majority of the program's macros.

Macroscope translates macros, conditional compilation, and include files into equivalent Ivy constructs. The latter two cases are straightforward, since Ivy supports compile-time conditionals and modules. Macros, however, are quite difficult to handle, since they often consist of arbitrary sequences of tokens. In such cases, Macroscope translates the tokens to complete syntactic units using a variety of heuristics. It understands the entire CPP language, including token pasting, stringization, recursive macros, and varargs macros. In some cases, it may make a construct less general (in order to convert it to a complete syntactic unit); these cases are called imperfect translations. For example, if a macro expands to an identifier that is used sometimes as a variable name and sometimes as a type name, then Macroscope will generate two different Ivy macros for the one CPP macro. However, it will always produce Ivy code that is equivalent to the original CPP code. Additionally, Macroscope's output is readable by humans and extremely similar to the original input.

Program	Lines	Imperfect macro translations	Imperfect #ifdef translations
gzip	7,324	7 (0.5%)	18 (2.0%)
rcs	17,178	10 (0.6%)	59 (5.8%)
OpenSSH	55,153	10 (0.1%)	41 (1.9%)
Linux 2.6.10	163,154	88 (0.6%)	62 (2.7%)

Table 1: Benchmarks demonstrating the feasibility of translating CPP code to Ivy. In the case of Linux, only a minimally configured kernel was translated. An imperfect translation is any construct that is not a nearly token-for-token translation of the C preprocessor code. Imperfect translations nevertheless produce correct code.

Macroscope's execution is divided into two phases: expansion and extraction. In the expansion phase, macros, conditionals, and include files are rewritten to C code as they would be by CPP. Macroscope keeps a record of each expansion. Next, the code is parsed into a syntax tree using a standard C parser. Afterwards, the extraction phase backs out each expansion in reverse order using the expansion records. To extract a given construct, Macroscope identifies the lowest syntax tree node that encompasses all of the tokens from the expansion record. This node is replaced with an Ivy construct that closely resembles the original CPP construct that was expanded, using several heuristics that allow us to generate good Ivy constructs. Every Ivy macro that Macroscope generates is built from an entire node in the syntax tree, which ensures that Ivy macros are complete syntactic units. This feature is crucial, since it allows Ivy macros to be parsed and analyzed as is.

We have tested Macroscope on a set of open source programs that we believe is representative of the systems programs that users will need to translate. Our largest test case is a minimally configured Linux 2.6.10 kernel. We also applied Macroscope to `gzip`, `rcs`, and `OpenSSH`. Table 1 shows the results for these benchmarks. Imperfect translations may be the result of Macroscope generating an Ivy construct that is less general than a given CPP construct, which is undesirable but sometimes necessary. Imperfect translations nevertheless result in correct Ivy code. Based on these results,

we believe that eliminating the preprocessor with Macroscope is completely feasible.

To demonstrate that Ivy code is not difficult to refactor, we have developed a proof-of-concept refactoring tool that operates on the code produced by Macroscope. We have applied the tool to the Ivy version of `gzip` 1.2.4, which contains a well-known buffer overflow vulnerability involving the `strcpy` function. The tool is designed to fix `strcpy`-based buffer overflows. It replaces each call to `strcpy` with a safer version that will not overflow its destination buffer. This safer function requires that the size of the destination buffer be passed as an argument. The refactoring tool attempts to infer the size of the buffer statically. If it fails, the user must supply the size argument manually. When we refactored `gzip`, the tool automatically inferred the buffer size about 70% of the time. The static analysis that the tool uses is currently extremely simple, but in the future we hope to scale it up for use on much more sophisticated properties.

6 Related Work

A number of projects have attempted to craft a successor to C. For example, Cyclone [10] is a C-like language that is type-safe and that provides advanced language features in a systems programming environment. In addition, BitC [13] is a language that attempts to combine C's expressiveness with the rigor of a modern functional language; it includes extensive support for formal verification. Unfortunately, neither of these languages provides a reasonable evolutionary transition path for existing software, without which it is difficult to make a real-world impact. BitC and Ivy could co-exist as they both follow the existing C binary interface. Rewriting some parts that need formal verfication in BitC, while using Ivy for the rest, is a plausible combination. In addition, our extensions and refactorings will allow programmers to customize Ivy for specific projects and to incorporate future language features, neither of which is supported by these alternatives.

Dawson Engler's research group has produced a number of program analysis tools, such as the MC system [9] and RacerX [5], both of which use static analysis to find bugs in large software systems. These projects have all been successful in uncovering serious bugs in real code; however, in order to scale to such large systems, they must make potentially unsound assumptions about properties of the code. Thus, these tools generate many false positives and occasional false negatives, making it impractical to incorporate them directly into the build process as we would like.

The SLAM project at Microsoft [1] is a program analysis tool that uses model checking to detect errors in Windows drivers. This tool provides even stronger guarantees about certain safety properties; however, it is currently used for individual drivers in isolation. A similar project at Microsoft, ESP [4], also can check that a program adheres to a given protocol. It is more scalable, but uses a weaker form of path sensitivity than SLAM. Nevertheless, we are encouraged by the results of MC, SLAM, and ESP, and we may use them as the basis for extensions and refactorings to check interface compatibility.

Finally, projects such as CCured [3] and Linux's Sparse [12] analyze existing software to improve memory safety and find defects. As mentioned earlier, these systems can be better implemented as part of our framework: the inference portion becomes a refactoring, and the static checks (as well as any corresponding run-time checks) become an extension.

7 Conclusion

The C language has a long and venerable history. Even today, despite its flaws, most systems software is written in C. However, in recent years, the flexibility of C has proven to be a liability as system designers focus more on reliability and security. We have proposed a new programming platform, Ivy, that provides the features of a modern, safe language along with an evolutionary path that will allow us to bring existing code up to date. Program transformations, called refactorings, are used to improve the safety and security of legacy code, and language extensions perform the necessary compile-time checks on refactored code. An initial translation step, which eliminates CPP and is mostly automatic, moves legacy code onto the platform, where the power of static analysis, refactoring and extensions can be fully applied. We hope Ivy will serve as a safe, modern platform for future systems research.

Acknowledgements: Thanks to George Necula, Rob von Behren and David Gay (of Intel) for their help on this project.

References

[1] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. *Lecture Notes in Computer Science*, 2057:103–122, 2001.

[2] H.-J. Boehm. Threads cannot be implemented as a library. Technical Report HPL-2004-209, Hewlett Packard, 2004.

[3] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 2003.

[4] M. Das, S. Lerner, and M. Seigle. Esp: Path-sensitive program verification in polynomial time. In *PLDI '02: Proceedings of*

the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, 2002.

[5] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 237–252. ACM Press, 2003.

[6] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 338–349. ACM Press, 2003.

[7] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 192–203, Atlanta, Georgia, May 1–4, 1999.

[8] S. N. Freund and S. Qadeer. Checking concise specifications for multi-threaded software. *Journal of Object Technology*, 3(6):81–101, 2004.

[9] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 69–82, 2002.

[10] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, June 2002.

[11] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *POPL '02: Proceedings of Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[12] D. Searls. Linus & the Lunatics, Part I. *Linux Journal*, November 2004. `http://www.linuxjournal.com/article/7272`.

[13] J. Shapiro, S. Sridhar, S. Doerrie, M. Miller, and E. Northup. The BitC language specification. `http://www.coyotos.org/docs/bitc-spec/bitc-spec.html`.

[14] H. Xi and F. Pfenning. Dependent types in practical programming. In ACM, editor, *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of Programming Languages, January 20–22, 1999, San Antonio, TX*, ACM SIGPLAN Notices, pages 214–227, New York, NY, USA, 1999. ACM Press.

Broad New OS Research: Challenges and Opportunities

Galen C. Hunt[1], James R. Larus[1], David Tarditi[1], and Ted Wobber[2]
[1]*Microsoft Research Redmond, Redmond, WA 98052, USA*
[2]*Microsoft Research Silicon Valley, Mountain View, CA 94043, USA*
http://research.microsoft.com/os/singularity

Abstract

Contemporary software systems are beset by problems that create challenges and opportunities for broad new OS research. To illustrate, we describe five areas where broad OS research could significantly improve the current user experience. These areas are dependability, security, system configuration, system extension, and multi-processor programming. In each area we explore how contemporary systems fall short. Where we have thought of possible solutions, we offer directions for future research.

To prove our point that opportunities for new OS research exist, we describe Singularity, a research project at Microsoft Research. Singularity is a new operating system designed to explore solutions to four of the challenges we have identified. Singularity incorporates three specific design decisions in order to increase system dependability and improve system security, configuration, and extension. These design decisions include the adoption of an abstract instruction set as part of the system binary interface, a unified extension architecture for both the OS and applications, and a first-class application abstraction.

1. Introduction

The products of forty years of OS research are sitting in everyone's desktop computer, cell phone, car, etc.—and it is not a pretty picture. Modern software systems are—broadly speaking---complex, insecure, unpredictable, prone to failure, hard to use, and difficult to maintain. Part of the difficult is that good software is hard to write, but in the past decade, this problem and more specific shortcomings in systems have been greatly exacerbated by increased networking and embedded systems, which placed new demands that existing architectures struggled to meet. These problems will not have simple solutions, but the changes must be pervasive, starting at the bottom of the software stack, in the operating system.

Unfortunately, as the emergence of the Internet exacerbated problems in conventional systems, the research community turned its attention from broad OS research to focus on incremental improvements or new areas such as distributed systems [17].

Without OS solutions, others stepped into the void by devising partial, application-level solutions to these problems. Consider, for example, the problem of isolating code from potentially untrusted sources. Applications and programming language runtimes have tried to supplant inadequate OS security with partially redundant and complex security abstractions using stack walking and code signing [12][24]. Others have attempted to solve this problem by replicating entire operating systems in virtual machine monitors for each security domain [11]. While the engineering is admirable, one wonders if the OS could provide a more integrated solution.

The remainder of this paper has three parts. Section 2 suggests example areas in which OS research could make operating systems work significantly better for most users. We offer these areas as evidence of opportunity, not as an exhaustive research agenda. Section 3 describes work in the Singularity project to address some of these areas. Finally, Section 4 summarizes the challenges and opportunities for broad new OS research and draw conclusions.

2. Opportunities for OS Research

To suggest the many opportunities for OS research, we list five areas in need of new ideas and abstractions: dependability, security, system configuration, system extension, and multiple processor programming. This list is intended to be illustrative, not exhaustive.

2.1 Dependability

A system is *dependable* if it behaves predictably and reliably; in other words, if its behavior consistently conforms to an understandable and useful model. A system's *perceived dependability* is a function of both user expectation and actual system behavior.

Unfortunately, the perceived dependability of contemporary software systems is low, particularly in the eyes of non-technical users [15].[1] Partially this results from raw software failures. However, it also results from unpredictable system behavior.

Broadly speaking, the owner of a modern PC encounters frequent unexpected behaviors. By contrast, most modern cars are considered quite dependable by their users; this despite the fact that cars can require as much

[1] Data from security advisories suggest that no contemporary system, either commercial or open source, has a monopoly on dependability problems [20].

as one hour of maintenance for every one hundred hours of usage.[2] We claim that modern cars are considered dependable because they have an easily understood operation model consisting of regular fueling, regular oil changes, regular maintenance, and basically predictable, uninterrupted usage the rest of the time.

No open, general purpose software system can make a similar claim. They all must be patched frequently and regularly to fix flaws that open the system to malicious attack. They all can fail in ways that are inexplicable and unpredictable to ordinary users. Many of these users are afraid to change their system in even the slightest way, for fear of breaking them.

2.2 Security

Contemporary OS security systems were designed to protect users of a system against each other and to protect the OS from errant programs. These security architectures were developed in the quaint past when code came from trusted sources and networks mostly connected us with our friends and colleagues. In today's connected world, users and computers are surrounded by unscrupulous advertisers, petty criminals, and increasingly organized crime. In this world in which executable code can and does come from anywhere, the OS needs to protect user and system resources from potentially hostile code that a user runs either intentionally or unintentionally. This is a very hard problem given that desired code may do useful work!

To bring code into an OS security model, there must be a basic OS abstraction that represents the identity of code. The abstraction should also capture the provenance of the code as well as provide a means for checking code integrity. Once code is identifiable, we can imagine enforcing security policy pertaining to it.

Code identity alone, however, is not sufficient. Software components interact in exceedingly complex ways, and many such interactions are security-relevant. We can expect the next generation of attacks to exploit unplanned and unprotected interactions between software components. There is fertile ground for research in understanding how to prevent such attacks by design.

The Java [12] and Common Language Infrastructure (CLI)[3] [24] programming environments have explored some of these issues. However, the security models in these systems are complex and largely separate from OS models.

[2] An oil change (1 hour) every 5,000 miles (100 hours at 50 miles/hour) is typical and does not take into account other preventive maintenance, which typically takes a car out of commission for an entire day.

[3] Microsoft's commercial implementation of the CLI is known as the Common Language Runtime (CLR). The CLR is the core of Microsoft's .NET Framework.

2.3 System Configuration

Contemporary operating systems contain abstractions for many components of modern applications, such as processes, threads, and shared libraries, but applications and their dependencies are only informally characterized. Lacking a strong concept of an application's complete configuration, the OS has no mechanisms to guarantee the integrity or provenance of an application. A system is only as stable as its most fragile component, which cannot be identified in current systems; systems which provide no easy way to distinguish application components intermixed in file systems and configuration registries.

Consider, for example, the case of applications colliding in their usage of shared spaces such as file systems or configuration registries. The installation of one application may corrupt or irreversibly alter the configuration of another via changes to a file or registry. The "DLL Hell" problem in Windows systems occurs when one application overwrites a common shared library with a version incompatible with an existing application. Similar problems can occur when an application overwrites configuration information mapping from document extensions to applications. To compensate for the absence of OS managed applications, users resort to ad-hoc application isolation techniques, such as jails [14] or virtual machine monitors, such as VMware [9] and Xen [3].

2.4 System Extension

Since no monolithic system can satisfy all users, most complex software lets users load code to extend functionality. Dynamically loaded extensions are found as widely as device drivers in kernels and spelling checkers in word processors. Whether in the OS or an application, most extensions are loaded directly into a host address space with no hard interface, protection boundary, or clear distinction between host and extension code. Extension through in-process code loading appears flexible and attractive, but due to a lack of isolation, extensions are a major source of software reliability and security problems. For example, faulty device drivers cause a large fraction of Windows and Linux failures [22].

A number of OS research efforts, including Exokernel [13], SPIN [5], VINO [21], and Nooks [22] have sought safer OS extension without addressing the more general problem of application extension. Pragmatically, each of these systems provided domain-specific models for OS extensions. Software fault isolation (SFI) [23], one of the few research efforts to consider application extension, limits an extension to a subset of an application's address space. However, the overhead for SFI is quite high and still exposes published data structures to corruption by the extension.

In Section 3.1.2, we will describe research in the Singularity system to create a unified extension architecture for both the operation system and applications.

2.5 Multi-processor Programming

Thanks to the physical constraints of semiconductor device scaling, it has become easier to replicate processors than to increase processor speed. Over the next decade the number of processing cores per chip could double every 18-24 months. Processing cores are replicating not only on CPUs, but in peripheral devices as well. Notwithstanding recent work on scheduling algorithms for multi-core CPUs [10] and programming GPUs [6], there are research opportunities to create new abstractions for programming large numbers of processors and to treat the non-CPU processors found in graphics, network, and storage devices as first-class compute resources.

3. Singularity

Singularity is a Microsoft Research project to develop techniques and tools for building dependable systems that address the challenges faced by contemporary software systems. Singularity is approaching these challenges by simultaneously pushing the state of the art in operating systems, run-time systems, programming languages, and programming tools—the foundation on which software is built. The Singularity OS is first and foremost a research system. Singularity strives for minimalism and design clarity, and makes extensive use of modern languages and tools.

By plan, performance is secondary to other research objectives such as security, dependability, and soundness of design. However, in places where we believe performance is central to the research challenge, such as streamlining cross-process communication, we strive for high performance solutions that also meet the other objectives.

To increase our ability to conduct a broad new OS research agenda, we have forgone compatibility with previous operating systems. Our experience is that new abstractions are best developed in an environment free of contradictory legacy requirements and then ported to legacy environments when the abstractions have matured. We recognize that this is a calculated risk; in the longer term, we have made provisions to implement a virtual machine monitor in Singularity as legacy support becomes a requirement.

3.1 Design Choices

A key focus of Singularity research is improving system dependability. Singularity improves dependability by dramatically increasing the scope of sound verification techniques to detect sources of unexpected system behavior. To broaden the scope of sound verification techniques, Singularity fixes the behavior of system components as early as possible in lifetime of their code (see Figure 1). To lengthen the scope of sound verification techniques, Singularity constrains system organization and preserves metadata so that verification results can be applied even to late-bound composites.

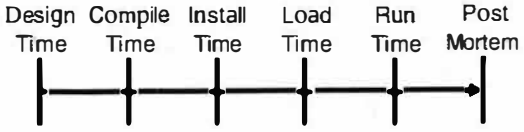


Figure 1. Code lifetime of a software component.

Singularity incorporates three key design choices to improve system dependability. These design choices are: an abstract instruction set as part of the system's application binary interface (ABI), a unified extension architecture, and a first-class application abstraction. The abstract instruction set provides the OS with a flexible layer of indirection between application code and a processor's instruction stream. The unified extension architecture enables rich, inexpensive, *and* safe interaction between system components. The application abstraction enables OS management of applications and integration of applications into the security model as security principles.

Early indications are that these design choices also have a positive impact on the challenges of system security, configuration, and extension. System security and configuration in Singularity are given much deeper treatment by Abadi *et al.* [1] and DeTreville [8], respectively.

3.1.1 Abstract Instruction Set

Singularity executables represent executable code in an abstract instruction set, called MSIL. MSIL is Microsoft's implementation of the ECMA Common Intermediate Language [25]. All third-party executables, including applications and device drivers, are delivered to Singularity as type-safe MSIL binaries.

Singularity requires that all user MSIL be type safe, which eliminates an entire class of programmer errors due to erroneous or malicious pointer arithmetic. Because Singularity controls the translation of MSIL into processor instructions, the OS retains the opportunity to insert trusted instruction sequences into the unprivileged, but verified, instruction stream. The abstract instruction set also opens new opportunities to dynamically adjust the trade-offs between security and performance, and it allows rigorous analysis and instrumentation of application code.

3.1.2 Unified Extension Architecture

Singularity provides one extension architecture for the operating system *and* applications. Like previous micro-kernels [2][16][18], Singularity incorporates a

process-based extension model. Singularity, however, assumes a more aggressive closed-process architecture for both OS and application extensions.

Singularity processes are closed worlds in two regards. First, Singularity disallows shared memory between processes; Singularity processes exchange data exclusively through messages, which are visible to only one process at a time. Second, once execution begins within a process, no new code may be added to the process. Singularity disallows both loading of new code modules and generation of new code into an existing process.

Any OS or application extension code can be loaded only into a child process, separated by a strong isolation boundary. Communication between host and extension across the process isolation boundary is restricted to verified message-passing *channels*. Channels are strongly typed with *contracts*. All cross-channel interactions and contracts are statically verified using a technique called *conformance checking* [7]. Conformance checking guarantees that a contract is fully specified, that two parties communicating through a contract will not deadlock, and that neither party will receive an unexpected message.

By disallowing dynamic loading of new code into a process, Singularity processes become a closed world in which analysis tools can make sound assumptions about process states, invariants, and valid state transitions. The closed-world extension architecture opens new opportunities for static analysis and optimization.

3.1.3 Application Abstraction

Singularity raises the notion of an application to a first-class OS abstraction. Applications have security identities and signed manifests declaring their constituent components. Installation, maintenance, and removal of applications are all operations controlled by the OS.

Applications are strongly isolated. Access to shared resources—including other applications—is mediated through the Singularity security model. The security model uses code identity and component relationships in access control checks [1].

The application abstraction is recursively applied to the OS itself, with the kernel and other OS components described by manifests. Manifests form the roots of a metadata infrastructure that enables introspection across the entire system –both applications and operating system. Through this metadata it should be possible, for example, to examine an offline Singularity system image and determine if it has the necessary components and configuration to run on a specific hardware configuration or host a specific application. A specific Singularity system as represented by an installation image then becomes a *self-describing artifact*, not just a collection of bits accumulated with at best an anecdotal history.

Most operating systems install and uninstall applications through imperative updates to mutable configuration information held in the file system and in configuration registries. We expect to extend Singularity's application abstraction to support a declarative form of configuration for a whole system, which we expect will eliminate whole classes of system misconfiguration [8].

3.2 Singularity Architecture

Singularity is a type-safe OS. Where traditional operating systems present untyped memory and the hardware instruction set to applications, Singularity replaces these with the abstractions of typed memory and an abstract instruction set, in the form of type-safe MSIL.

Singularity relies on type-safety and control of the translation of the abstract instruction set to machine code to enforce system protection boundaries. This allows faster and more efficient process-to-kernel context switches and communication between processes.

At the heart of the system is a trusted computing base (TCB), see Figure 2. The Singularity TCB is composed of the kernel proper, trusted runtime code, and MSIL translators. The TCB maintains security policies and guarantees that no untrusted or unverified instructions ever execute. The TCB ensures process integrity by providing isolated object spaces for processes and constraining IPC communication to contract-conforming channels.

Most of the TCB is written in Sing#, an extension of C# with specifications on objects and conformance-checked channels. The object specifications come from Spec# [4], which extends C# with pre-conditions and post-conditions on methods, and invariants on class variables. An implementation conforms to a contract if it only sends or receives messages over the channels those message described in the channel and all channel-visible state changes conform to the state machine in the channel contract.

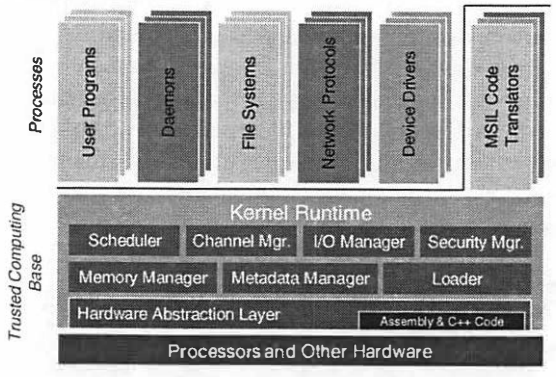


Figure 2. Singularity Architecture.

Portions of the TCB, including the per-process garbage collectors (GCs), are written in unsafe C#. At the

bottom of the system, a small body of C++ and assembly code provides the lowest portions of the hardware abstraction layer (HAL). Spec# and C# codes are emitted as MSIL and translated to hardware instructions. The C++ code is compiled directly to the hardware instruction set.

All third-party binaries, including applications, extensions, and device drivers, are delivered to Singularity as type-safe MSIL binaries. Each process receives its own memory pages, but type safety and garbage collection guarantee that no process can hold pointers to any page it does not rightfully own. As a result, most of the system, including third-party code, can run in the same address space and hardware protection domain as the kernel.

MSIL binaries may be translated into hardware instruction streams at load or install time based on metadata in the application manifest. Caching of hardware instruction streams is invisible to both applications and users.

The Singularity kernel integrates some of the runtime services of the CLI with traditional kernel-based OS services such as scheduling, IPC, and I/O management. By redrawing the line between the runtime and the kernel, Singularity eliminates redundancies in resource management and security policy. The runtime also enjoys access to kernel features, such as direct access to the processor's MMU.

Singularity's implementation of CLI features is factored to minimize code in the trusted computing base. Code translators reside in processes outside the kernel and convert MSIL into verified hardware instruction streams. The loader caches hardware instruction streams and maps them into processes. The memory manager includes the GC and its accompanying facilities such as the GC write barrier. The metadata manager acts as a repository for traditional CLI code metadata, such as type information required for garbage collection. The metadata manager also coordinates information related to the application abstraction and application manifests.

The Singularity architecture supports multiple MSIL code translators. Individual translators may generate qualitatively different code from the same input. For example, one translator might optimize for performance while another may optimize for security by insert security automata [19] into the code. In the future, additional translators might target secondary processors such as GPUs.

3.3 Project Status

The Singularity system has been under design and development for a little over a year. Although still a work in progress, Singularity is now a recognizable operating system with threads, processes, channels, an I/O subsystem, device drivers, a TCP/IP network stack, a file system, a base CLI class library and runtime, and a kernel debugger. Singularity boots on PC hardware using the NVIDIA nForce4 chipset and under the Virtual PC VMM. Notable missing features include a GUI and virtual memory paging. The first version of the application abstraction work is coded, but has not yet been integrated with the rest of the system.

Over the next year, we intend to deploy the Singularity system and a small set of applications into the homes of approximately 50 researchers as a home service appliance. Our test deployment will target non-traditional applications, in particular, applications where the service appliance hosts services provided and managed by multiple third parties. A key objective of the deployment is to measure dependability of the current architecture and to experiment with the application abstraction to automate system configuration.

4. Conclusions

The world needs broad operating system research. Dependability, security, system configuration, system extension, and multi-processor programming illustrate areas were contemporary operating systems have failed to meet the software challenges of the modern computing environment.

The OS research community, in collaboration with researchers from the computer architecture, programming languages, and software tools communities, are well positioned to provide innovative solutions to today's software challenges. If the research community fails to take up this challenge, practitioners will likely provide incomplete solutions developed under competitive duress; the outcome is not likely to be a happy one.

Contemporary operating systems, both proprietary and open source, are constrained by backward compatibility and are unlikely to make the radical changes necessary to improve a typical user's computing experience without clear research guidance. A generation of orthodoxy has led software systems to this unsatisfying state.

We believe the OS research community should embrace this opportunity. We recognize that such opportunity does not come without risk. Many nice research OS abstractions have fallen by the wayside. However, as user dissatisfaction with the status quo continues to rise, unique opportunities may arise for either new operating systems or adoption of new OS abstractions within existing systems.

For our part, the Singularity project is responding to this opportunity by re-examining the fundamental abstractions of software systems through adoption of three design choices: an abstract instruction set, a unified extension architecture, and a first-class application abstraction.

5. Acknowledgements

We thank Martín Abadi, Mark Aiken, Paul Barham, John DeTreville, Orion Hodson, Mike Jones, Nick Murphy, and Ben Zorn for their help preparing this paper. We also thank the anonymous referees for valuable suggestion to improve the content and presentation of this paper.

References

[1] M. Abadi, A. Birrell, and T. Wobber. Access Control in a World of Software Diversity. *Proc. of Hot OS X: The 10th Workshop on Hot Topics in Operating Systems,* June 2005.

[2] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. *Summer USENIX Conference*, pp. 93-112, 1986.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield. Xen and the Art of Virtualization. *Proc. of the 19th ACM Symposium on Operating Systems Principles*, pp. 164-177, 2003.

[4] M. Barnett, K. R. M. Leino, and W. Schulte, The Spec# Programming System: An Overview. *Proc. of Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, 2004.

[5] B. N. Bershad, S. Savage, P. Pardyak, E.G. Sirer, M.E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. *Proc. of the 15th Symposium on Operating Systems Principles*, pp. 267-283, 1995.

[6] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brooks for GPUs: stream computing on graphics hardware. *Proc. of the 2004 SIGGRAPH Conference*, pp. 777-786, 2004.

[7] S. Chaki, S. K. Rajamani, and J. Rehof, Types as Models: Model Checking Message-Passing Programs. *Proc. of the 29th ACM Symposium on Principles of Programming Languages*. pp. 45-57, 2002.

[8] J. DeTreville. Making system configuration more declarative. *Proc. of Hot OS X: The 10th Workshop on Hot Topics in Operating Systems*, June 2005.

[9] S. Devine, E. Bugnion, and M. Rosenblum. Virtualization system include a virtual machine monitor for a computer with a segmented architecture. *US Patent*, 6397242, 1998.

[10] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Implementing an OS Scheduler for Multithreaded Chip Multiprocessors. *Work-in-Progress Reports, 6th Symposium on Operating Systems Design and Implementation*, 2004.

[11] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, D. Boneh. Terra: A Virtual-Machine Based Platform for Trusted Computing. *Proc. of the 19th ACM Symposium on Operating Systems Principles*, pp. 193-206. 2003.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java language specification.* Addison-Wesley, 2000.

[13] M. Kaashoek, D.R. Engler, G.R. Ganger, H.M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems, *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pp. 52-65. 1997.

[14] P.H.Kamp and R.N.M. Watson, Jails: Confining the omnipotent root. *SANE 2000.* May 2000.

[15] C. Kaner and D.L. Pels, *Bad Software: What to Do When Software Fails.* John Wiley & Sons, 1998.

[16] J. Liedtke. Toward real μ-kernels. *Communications of the ACM*, 39(9):70-77, September 1996.

[17] R. Pike. Systems Software Research is Irrelevant. *Invited talk*, University of Utah, February 2000.

[18] M. Rozier, A. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. CHORUS distributed operating system. *Computing Systems*, 1(4):305-370, 1988.

[19] Schneider, F.B. Enforceable Security Policies. *ACM Transactions on Information and System Security* (TISSEC), 3 (1). 30-50, 2000.

[20] Secunia, *Statistics of released advisories by project*, http://secunia.com/product, 2005.

[21] M. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. *Proc. of the 2nd Symposium on Operating Systems Design and Implementation*, pp. 213-228, 1996.

[22] M. Swift. B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems, *ACM Transactions on Computer Systems*, 22(4), Nov. 2004.

[23] R. Wahbe, S. Lucco, T.E. Anderson, and S. Graham. Efficient Software-Based Fault Isolation. *Proc. of the 14th ACM Symposium on Operating Systems Principles*, pp. 203--216 1993.

[24] *Common Language Infrastructure (CLI): Partition I: Architecture*, ISO/IEC 23271:2003.

[25] *Common Language Infrastructure (CLI): Partition II: CIL Instruction Set*, ISO/IEC 23271:2003.

`patch (1)` Considered Harmful

Marc E. Fiuczynski
Princeton University

Robert Grimm
New York University

Yvonne Coady
University of Victoria

David Walker
Princeton University

Abstract

Linux is increasingly used to power everything from embedded devices to supercomputers. Developers of such systems often start with a mainline kernel from `kernel.org` and then apply patches for their application domain. Many of these patches represent *crosscutting concerns* in that they do not fit within a single program module and are scattered throughout the kernel sources--easily affecting over a hundred files. It requires nontrivial effort to maintain such a crosscutting patch, even across minor kernel upgrades due to the variability of the kernel proper. Moreover, it is a significant challenge to ensure the kernel's correctness when integrating multiple crosscutting concerns. To make matters worse, developers use simple code merging tools that directly manipulate source file lines instead of relying on a lexical, grammatical, or semantic level of abstraction. The result is that patch maintenance is extremely time consuming and error prone. In this paper, we propose a new tool, called `c4`, designed to help manipulate patches at the level of their abstract syntax and semantics. We believe our approach will simplify the management of OS variations and thereby improve OS evolution.

1 Introduction

Over the past years open source operating systems, particularly Linux, have experienced tremendous growth. Industry and governments alike are relying upon such software to reduce the cost and time-to-market of developing WiFi routers, cell phones, and telecommunications equipment and of running services on specialized servers, clusters, and high-performance supercomputers. One important benefit of using Linux for these systems is that developers have access to all kernel sources and can easily create variants that are directly tailored for their application domains. As such, Linux also is an attractive platform for OS research, as it offers the potential for a speedy technology transfer.

Major variants to a mainline Linux kernel are typically represented in terms of higher-level extensions that are implemented through so-called patch sets or, simply, patches. For example, embedded systems require changes that reduce the kernel's memory footprint (e.g., Linux-Tiny [21] or uCLinux [32]), desktops require strong security mechanisms that reduce the impact of viruses and worms (e.g., LSM [20]), time-shared servers require resource management subsystems to isolate users from each other (e.g., VServer [23] or CKRM [4]), and super computers require special resource management modifications that scale the OS to a large number of components (e.g., CPUSETS [9]). Many of these kernel extensions do not fit within a single source file and are scattered throughout the kernel sources. As shown in the following table, each extension can easily cover a hundred existing kernel files, even though it represents a logical unit, expressing a single *crosscutting concern*:

Patch	New Files	Modified Kernel Files
Nooks [30]	68	108
CKRM [4]	48	53
LSM [20]	123	85
Kernel Probes [17]	13	20
LTT [22]	9	71
VServer [23]	40	211
Linux-Tiny [21]	7	142
CPUSETS [9]	1	3
ALSA [1]	200	540
LLA [24]	1	39

It requires non-trivial effort to maintain even a small crosscutting extension between minor kernel upgrades due to the variability of the kernel proper. Moreover, it is a tremendous challenge to ensure the kernel's correctness when integrating multiple crosscutting kernel extensions, as even for the small number of patch sets shown in the above table there is significant overlap in the files affected by the different extensions. To make matters worse, developers currently use simple code merging tools (e.g., `diff` and `patch`), which are limited to indicating conflicts based upon textual comparison. Experience with maintaining a variant of the Linux kernel for PlanetLab (which contains several major variants to the mainline code base) as well as anecdotal evidence from the Linux and OS research communities suggest that this approach is error prone and time consuming.

Developers wishing to merge their kernel extensions into the mainline code base must repeatedly go through this process, because any non-trivial change to Linux's architecture takes time to be reviewed and accepted. Anecdotal evidence (e.g., LSM, LTT, ALSA) suggests that it may take anywhere from one to three years before a crosscutting kernel extension is fully integrated into the mainline kernel.

This leads to a natural selection of kernel extension developers: those that are persistent and those that are not. While this natural selection weeds out the weak, it also eliminates strong work done by members of the OS research community. For example, the Nooks [30] project for recoverable Linux device drivers has garnered best paper awards at both SOSP and OSDI. Yet the work remains relegated to Linux 2.4.18, which was the kernel version at the start of Nooks in Feb. 2002. The problem is not laziness! Rather, with today's tools, it is simply too tedious to keep up with the changes that occur between even minor releases of Linux, e.g., from 2.4.18 to 2.4.19.

Our position is that a better method is needed—beyond `diff` and `patch`—that reduces the amount of work it takes to maintain and review a crosscutting kernel extension in Linux. The remainder of this paper describes our work towards such a solution: a *semantic patch system* called `c4` for CrossCutting C Code. A semantic patch basically amounts to a set of transformation rules that precisely specify the conditions under which changes need to be made and the means for rewriting the affected code. Its compact yet human readable form lets a community of developers easily understand and discuss a crosscutting kernel extension, thereby helping reduce the time and effort required to evolve the kernel.

2 Motivating Observations

We studied a number of patch files—LTT, Kernel Probes, LSM, and CKRM—that introduce new kernel extensions as well as patch files that update existing code in Linux. In general, the changes introduced by these patch files fall into three categories*:

- *Intraprocedural changes.* Modifications to the internal logic of an existing function. These changes may eliminate bugs, but they do not change the type of the function and normally do not change its original intended semantics.
- *Intramodule changes.* A coherent collection of modifications encapsulated within an existing module. These changes may modify many of the functions within a particular module, but they do not change the externally visible interface. Clients of the module do not need to change their code and usage patterns. We use the word "module" loosely here to mean any collection of components with a well-defined external interface including kernel subsystems. In particular, a module is not limited to a single kernel source file. For instance, changing all file operations to use ACLs rather than standard UNIX permissions would be an intramodule change.
- *Intermodule changes.* Modifications that change intermodule interfaces or the visible semantics of an existing interface in fundamental ways. For instance, modifications of a function's type or the field makeup of a non-ADT data structure (e.g., adding, deleting, or changing the type of a field) are intermodule changes.

Our preliminary analysis of patches that update existing kernel functionality shows that the bulk fall into the *intraprocedural* and *intermodule* changes categories with very few *intramodule* changes category. In contrast, patches that introduce new kernel extensions fall primarily into the *intramodule* and, to a lesser extent, the *intermodule* changes categories.

As the size and scale of a kernel modification moves from intraprocedural to intramodule to intermodule, it becomes more and more difficult to apply and maintain the modification using the low-level line-by-line diffs that result from `patch`. There are two important reasons for this. First, the more expansive intra- and intermodule changes almost always encapsulate a new *coherent* unit of functionality. However, patch sets provide the programmer with no help in understanding the new unit of functionality in isolation. Since patch sets do not even operate on the concrete syntax of the programming language, they are often syntactically invalid fragments of code. They may also contain free variables that can only be resolved when the patch is applied to the kernel. Consequently, it is impossible to assign patches any precise semantics separately from the modules they modify.

Second, as the scope of a kernel modification extends, it becomes more and more likely that the change will conflict with other changes being made simultaneously to the kernel. The difficult and error-prone part of maintaining patches is attempting to understand these conflicts. Often, spurious conflicts arise due to the fact that patches operate on a line-by-line basis without regard either to the basic syntax of the language or, more importantly, to its semantics. In other words, when different developers modify the same lines of a file, `patch` will signal a conflict and a programmer must analyze, understand and modify the code at that location, even though the changes may be semantically independent of one another. Due to the complete lack of any abstraction or semantic meaningfulness of patches, it is easy to make mistakes during this process.

*A fourth category comprises modifications to Makefiles etc., which we do not consider.

To illustrate these problems in more detail, consider the patch set for the Class-Based Kernel Resource Management (CKRM) project [4]. CKRM is a new kernel mechanism that provides differentiated services for shared system resources, including CPU time, tasks, memory, and disk I/O. The application of this patch set results in a new Linux kernel variant suitable for servers that require stronger resource usage guarantees than the egalitarian approach used by the unmodified mainline kernel.

The actual CKRM extension consists of a set of files that specify where "hunks" of code are applied by `patch`, identifying specific line numbers or relative offsets within specific files. For example, this portion of the CKRM patch:

```
--- a/kernel/sys.c Sat Sep 18 19:28:57 2004
+++ b/kernel/sys.c Tue Feb 01 22:03:15 2005
@@ -638,6 +642,9 @@
  else
     return -EPERM;
+
+ ckrm_cb_gid();
+
  return 0;
@@ -726,6 +733,8 @@
     current->suid = current->euid;
  current->fsuid = current->euid;
+
+ ckrm_cb_uid();
+
  return security_task_post_setuid(old_ruid,
```

instruments `kernel/sys.c` with calls to the ckrm_cb_gid()/uid() functions. Line numbers are represented as relative offsets indicated by *@@line-info@@*. Upon closer inspection of the patch we observe a pattern: all calls to ckrm_cb_gid()/uid() (6 in total) directly precede return statements.

The same pattern emerges for other kernel extensions, such as LSM and VServer, that hook themselves into specific Linux subsystems. Thus, composing several such kernel extensions or updating to a new release of the kernel may result in unnecessary patch conflicts. Such conflicts typically need to be resolved manually, which is clearly tedious. Section 3 presents our solution to this problem, which transforms these *intramodule changes* into *aspects* using aspect-oriented software development [2] (AOSD) techniques.

Intermodule changes often involve modifications to either a function's signature or the field makeup of a data structure. But changing a function's signature or deleting/changing a data structure's fields can have far-reaching consequences: it requires updating all modules that directly use them. Consequently, capturing such changes with `diff` and `patch` requires manually updating all dependent modules. This is prohibitive when the interface changes are in the kernel proper or in the generic device driver framework and trigger corresponding changes in specific device drivers—there might be hundreds.

The Bossa project [13, 27] encapsulates new functionality for Linux in a single component, where the component interface specifies rewrite rules to compose the code with the base program. The rewrite rules leverage temporal logic to describe execution points in the program. Lawall et al. [19] attack this problem at a different level, percolating interface changes throughout the Linux code base. Similar to our approach, this work builds on a kind of semantic patch, which relies on code rewriting rules to automate the task of updating dependent modules. While this appears promising, we observe that *intermodule changes* might be better handled by: (1) a systematic conversion of non-ADTs used across Linux subsystems to ADTs, thereby making further changes to them intramodule changes, and (2) using well-established interface versioning techniques such as Microsoft's Component Object Model (COM).

3 The c4 Semantic Patch Compiler

Recognizing that intramodule and intermodule changes are common to new kernel extensions for Linux, our approach is to make them part of the kernel's architecture by leveraging AOSD techniques. More specifically, our approach is to express these changes as semantic patches using aspects, which provide a language-supported methodology for integrating crosscutting concerns with a program. The benefits of aspects are twofold. First, they provide a well-defined specification of domain-specific features that is separate from baseline functionality, yet can be automatically integrated with the kernel. Second, we believe that aspects enable tools that perform automatic analysis of the implications of composing several crosscutting concerns and identify true semantic conflicts as opposed to the line-by-line conflicts identified by `patch`.

The main research questions raised by our approach are (1) how to extend C with aspects without impacting compatibility, readability, or performance and (2) how to automate the identification and resolution of conflicts between aspects. However, fully exploring these research questions requires building the corresponding tools. To reduce the required engineering effort, we are not implementing a self-contained C compiler for our AOSD-enhanced C language, called `c4` for CrossCutting C Code. Rather, we leverage existing platform support for C and rely on a pipeline that first invokes the C preprocessor, which resolves all # directives, then the `c4` compiler to translate aspect-enhanced code to plain C, and finally gcc, which performs traditional optimizations and code generation. To further reduce the engineering effort required for building `c4`, we are implement-

ing c4 on top of the xtc compiler framework [14, 15], which provides a toolkit for building extensible source-to-source transformers. In the rest of this section, we present the proposed aspect-oriented language enhancements to C by example and then discuss our approach to non-interference analysis for aspects.

3.1 The c4 Language

In c4, which is based on AspectC [8], aspects structure and modularize concerns that crosscut functions. Due to space constraints, we do not define the c4 language in detail. Rather, we illustrate the gist of its features on the example of instrumenting the kernel with calls to ckrm_cb_gid()/uid() *after* the *execution* of sys_setregid()/setreuid(), sys_setgid()/setuid(), and sys_setresgid()/setresuid(), respectively:

```
aspect (CKRM) {
     pointcut setuid() :
       execution(long sys_setreuid(..))  ||
       execution(long sys_setresuid(..)) ||
       execution(long sys_setuid(..));

     after setuid() { ckrm_cb_uid(); }

     pointcut setgid() :
       execution(long sys_setregid(..))  ||
       execution(long sys_setresgid(..)) ||
       execution(long sys_setgid(..));

     after setgid() { ckrm_cb_gid(); }
}
```

The *execution* keyword refers to principled points in the execution of a program called *join points* (e.g., sys_setreuid). A *pointcut* statement groups one or more join points, which can then be referenced by *advice* to define actions for these join points. In our example, we only use *after* advice, which indicates that the actions (the explicit C code) should be performed after the *execution* of the join points.

The aspect code thus structures the modifications to the mainline code, which are automatically merged, or *weaved*, with the appropriate C code by the c4 compiler. In contrast to the line-by-line patch shown in Section 2, the interaction with the kernel becomes explicit at the level of functions and parameters involved; hence, code becomes more amenable to semantic analysis and developers can reason about any interactions at a higher level. Previous work has shown that this reduces the complexity of crosscutting concerns [6, 7, 28].

Note that the c4 language is richer than suggested by this example. In particular, it supports not only *after*, but also *before* and *around*, with the latter replacing an existing mainline function. Coady describes this in further detail for AspectC [8], upon which c4 is based. Further note that we aim to reduce developers' exposure to c4 as much as possible. In particular, we are exploring how to support simple annotations of the form aspect(Name){...}, which can be added inline at the beginning or end of system functions and are then automatically extracted and converted into fully-featured aspects by the c4 compiler.

3.2 Program Analysis

Our initial research goal is to support the *syntactic* separation of crosscutting concerns through aspects. On their own, syntactic separation and automatic weaving of crosscutting concerns free developers from many low-level, time-consuming, and error-prone details of maintaining and applying kernel patches. However, in addition to supporting syntactic separation of crosscutting concerns, we are also targeting *semantic* separation through the detection of interference between concerns. When two concerns are semantically separate, the execution of one concern is guaranteed not to change the execution behavior of the other. For instance, semantically separate concerns do not mutate shared data structures either directly or indirectly through a series of function calls. Semantically separate concerns are of critical importance in large systems such as Linux, in which multiple developers work independently on their own system extensions. When concerns are semantically separate, these independent developers need not coordinate their work, analyze the code of the other developers, or even be aware that other projects are being developed. By definition, the work of one developer does not interfere with the other.

In addition to separating multiple "after-the-fact" concerns, it is useful to determine the degree to which a particular concern is separate from, or, conversely, interferes with, the mainline code. If a developer can prove, via an automated program analysis, that their concern does not interfere with the mainline code, then owners of the mainline are much more likely to integrate it into their system. Even if the owners themselves will not integrate the new concern, users will be less hesitant to download and apply the non-interfering kernel extension. We believe that analysis of noninterference properties of aspects can greatly speed technology transfer and integration of new ideas into Linux (and other open source software).

We have begun to investigate how to design a static program analysis that will detect whether a new concern interferes with the mainline computation [10] or with another, existing concern [3]. This analysis makes use of previous work developed by programming language and security researchers on detecting and enforcing data integrity properties via information flow analysis. Our analysis is designed as a form of type-and-effect system that separates state into different logical protection domains, with one protection domain for each concern and one domain for the mainline computation. The analysis is designed to detect situations, in which code from one

domain mutates state in another, either directly or indirectly through a series of function calls. We have formally proven a powerful non-interference result for our analysis.

While an important step, there still are considerable challenges to using this analysis in the context of C and the Linux kernel. A first step for this research will be to refactor crosscutting concerns in Linux and to analyze the degree to which various concerns really are separate from one another and the mainline kernel. This experience will be crucial in refining the theoretical analysis and in understanding the specific noninterference properties that will be useful (and possible) to specify and enforce. The next step will be to extend `c4` with a system of annotations that let developers specify their noninterference and semantic separation requirements. Even without an analysis, the annotation system will be useful as a systematic form of documentation of developer intentions and requirements. The last step is to implement the analysis itself and test it on kernel extensions in Linux.

4 Related Work

Both IBM and Microsoft recently announced their intentions on using AOSD to improve the evolvability of complex software systems [18, 31]. Our work differs from these industry initiatives in three important aspects. First, we are investigating aspect-oriented programming within C as opposed to existing efforts on C++, C#, or Java. Consequently, our work directly applies to a large, existing code base. Second, we are specifically targeting the software architecture of a major open source operating system, which provides us with an opportunity to address a real-world problem faced by many organizations. Finally, we plan to develop formal semantics for reasoning about aspect-oriented technology in this domain, and use this formalization to develop program analysis tools to further aid systems programmers in general.

AOSD techniques have been previously applied to operating systems. Both Coady et al. [8] and Spinczyk et al. [25] demonstrate that concerns that crosscut traditional layers in OS structure can be cleanly defined and applied using aspects.

Over the last several years, a number of researchers have begun to build semantic foundations for aspect-oriented programming [34, 11, 16, 26, 5, 33]. This foundational, theoretical work provides a starting point for analyzing the properties of aspect-oriented programs, developing principled new programming features, and deriving useful program analyses. We plan to exploit our knowledge of and experience with these semantic foundations and type-based analyses as we develop the `c4` language.

Recently, programming language researchers have also begun to try to understand and analyze interactions between separate concerns. For instance, Bauer et al. [3] introduces a theoretical language that includes several different ways for combining concerns and a type system for detecting when concerns apply to the same program points. In similar work, Douence et al. [12] analyze aspects defined by recursion together with parallel and sequencing combinators. They develop a number of formal laws for reasoning about their combinators and an algorithm that is able to detect aspect independence. These proposals present interesting techniques for detecting interference, but it appears that additional reasoning facilities will be required for analyzing crosscutting concerns in the Linux kernel, as many of the "separate" concerns actually reference the same program points. We believe that more recent work by Rinard [29] and Dantas [10], which analyzes aspect code to determine its memory effects, will help solve this problem.

5 Summary

Our position is that current techniques for evolving operating systems are ineffective, since they solely operate at a line-by-line level. Our work introduces a semantic patch system based on *aspects* that offers the ability to more rapidly and seamlessly move from idea to design to implementation for new OS features. Aspects' inherent separation of code from an operating system's mainline eases the maintenance of crosscutting concerns, thus speeding up the technology transfer of a new kernel extension from early prototype, through multiple design iterations, to a mainlined feature of an operating system that continues to evolve.

References Cited

[1] Advanced Linux Sound Architecture. `www.alsa-project.org`.

[2] Aspect Oriented Software Development. `www.aosd.net`.

[3] L. Bauer, J. Ligatti, and D. Walker. Types and effects for non-interfering program monitors. In *International Symposium on Software Security*, pages 154–171, Tokyo, Japan, Nov. 2002.

[4] Class-Based Kernel Resource Management. `ckrm.sf.net`.

[5] C. Clifton and G. T. Leavens. Assistants and observers: A proposal for modular aspect-oriented reasoning. In *Foundations of Aspect Languages*, Apr. 2002.

[6] Y. Coady and G. Kiczales. Back to the future: A retroactive study of aspect evolution in operating system code. In *International Conference on Aspect-Oriented Software Development (AOSD)*, 2003.

[7] Y. Coady, G. Kiczales, A.Warfield, and M. Feeley. Brittle systems will break not bend: Can AOP help? In *10th ACM SIGOPS European Workshop on Operating Systems*, 2002.

[8] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, 2001.

[9] CPUSETS for Linux. `www.bullopensource.org/cpuset`.

[10] D. S. Dantas and D. Walker. Harmless advice. In *Workshop on Foundations of Object-oriented Languages*, Jan. 2005.

[11] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Third International Conference on Metalevel architectures and separation of crosscutting concerns*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186, Berlin, Sept. 2001. Springer Verlag.

[12] R. Douence, O. Motelet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Conference on Aspect-Oriented Software Development*, pages 141–150, Mar. 2004.

[13] R. Eberg, J. Lawall, M. Sudholt, G. Muller, and A.-F. L. Meur. On the automatic evolution of an os kernel using temporal logic and aop. In *Automated Software Engineering 2003 (ASE 2003)*, Oct. 2003.

[14] R. Grimm. Practical packrat parsing. Technical Report TR2004-854, New York University, Mar. 2004.

[15] R. Grimm. Systems need languages need systems! In *Proceedings of the 2nd ECOOP Workshop on Programming Languages and Operating Systems*, Glasgow, United Kingdom, July 2005.

[16] R. Jagadeesan, A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *European Conference on Object-Oriented Programming*, Darmstadt, Germany, July 2003.

[17] Kernel Probes. `www-124.ibm.com/linux/projects/kprobes`.

[18] G. Kiczales. The more the merrier. Software Development Magazine `www.sdmagazine.com/documents/s=8993/sdm0410g`, 2004.

[19] J. L. Lawall, G. Muller, and R. Urunuela. Tarantula: Killing driver bugs before they hatch. In *Fourth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, March 2005.

[20] Linux Security Modules. `lsm.immunix.org`.

[21] Linux-Tiny. `www.selenic.com/tiny-about`.

[22] Linux Trace Toolkit. `www.opersys.com/LTT`.

[23] Linux VServer Project. `linux-vserver.org`.

[24] Low Latency Audio. `www.linuxdj.com/audio/lad/resourceslatency.php`.

[25] D. Mahrenholz, O. Spinczyk, A. Gal, and W. Schröder-Preikschat. An aspect-oriented implementation of interrupt synchronization in the Pure operating system family. In *Proceedings of the 5th ECOOP Workshop on Object Orientation and Operating Systems*, Malaga, Spain, June 2002.

[26] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs. In G. T. Leavens and R. Cytron, editors, *Foundations of Aspect-Oriented Languages Workshop*, pages 17–25, Apr. 2002.

[27] G. Muller, J. Lawall, J.-M. Menaud, and M. Sudholt. Constructing component-based extension interfaces in legacy systems code. In *11th ACM SIGOPS European Workshop*, Sept. 2004.

[28] G.C. Murphy, R. J. Walker, and E. L. A. Baniassad. Evaluating emerging software development technologies: Lessons learned from assessing aspect-oriented programming. *IEEE Transactions on Software Engineering*, 25(4), 1999.

[29] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *Twelfth International Symposium on the Foundations of Software Engineering*, Nov. 2004.

[30] M. Swift. Nooks: Improving the reliability of commodity operating systems. `nooks.cs.washington.edu`.

[31] TheServerSide at AOSD 2004: Part Two. `www.theserverside.com/articles/article.tss?l=AOSD2004-2`, 2004.

[32] uCLinux. `www.uclinux.org`.

[33] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *ACM SIGPLAN International Conference on Functional Programming Languages*, Upsala, Sweden, Aug. 2003.

[34] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In G. T. Leavens and R. Cytron, editors, *Foundations of Aspect-Oriented Languages Workshop*, pages 17–25, Apr. 2002. Iowa State University University technical report 02-06.

WiDS: an Integrated Toolkit for Distributed System Development

Shiding Lin, Aimin Pan and Zheng Zhang
Microsoft Research Asia
{slin, aiminp, zzhang}@microsoft.com
Rui Guo[†]
Beijing University of Aeronautics and Astronautics
guorui@sei.buaa.edu.cn
Zhenyu Guo[†]
Tsinghua University
guozy03@mails.tsinghua.edu.cn

Abstract

Faced with a proliferation of distributed systems in research and production groups, we have devised the WiDS ecosystem of technologies to optimize the development and testing process for such systems. WiDS optimizes the process of developing an algorithm, testing its correctness in a debuggable environment, and testing its behavior at large scales in a distributed simulation. We have developed many distributed protocols and systems using WiDS, including a large-scale backup service that is robust enough to be deployed. We have also used WiDS to perform ultra-large scale (>1million instances) simulation of a production protocol. In this paper, we describe the principles and design of WiDS, share the lessons that we learned, and discuss on-going research that will further reduce programming and debugging difficulties of distributed systems.

1. Introduction

Research and development of distributed system has always been a tricky business. The process has many different stages, and each interdependent stage carries different requirements. The protocols must first be fully specified and proved. A correct implementation that follows is no trivial matter, as debugging a distributed system is a known hard problem. For the purpose of developing Internet-scale P2P systems [1][2][3], perhaps the most challenging is to fully understand any performance issues before the system is deployed.

To mitigate some of these difficulties, we find that a systematic approach is helpful. While the protocol specification, modeling, and proof remain too difficult to be incorporated in an integrated toolkit, we have united the rest of the processes in a single integrated toolkit called WiDS (*Wi*DS *i*mplements *D*istributed *S*ystem).

The general philosophy of WiDS can be summarized as "code once and run many ways". WiDS adopts an object-oriented and event-driven programming model, and provides a small and straightforward set of APIs to support message exchanges and timers. Once a distributed protocol is developed, it can be simulated within a single address space on a single machine for debugging purposes, simulated on a cluster of machines to understand its macro-behavior, or deployed and run in the real. Users work with the same code base across different development stages and link it to appropriate libraries accordingly.

We have researched and developed many of our protocols and systems using WiDS, including a large scale, distributed backup service [4] that is robust enough to be deployed in MSR-Asia this year. We have also done extensive testing for production code of a P2P protocol [5] of more than one million instances, using hundreds of clustered PCs. To our knowledge, this is the largest P2P simulation that has ever been attempted. While all these exercises have demonstrated the value of such an integrated toolkit, our experiences also point out more challenging research directions to further reduce programming difficulties as well as to improve the debugging process.

Section 2 gives an overview of WiDS. We summarize our experience of performing complete system development and large-scale testing in Section 3. We discuss several new research focuses in Section 4. Section 5 discusses related work, and we conclude in Section 6.

2. The WiDS Ecosystem

To serve as a generic ecosystem for distributed system development, WiDS needs to achieve several specific goals. First, there should be one single code base that is

[†] Work is done as intern in Microsoft Research Asia.

easily shared across different development stages. It is hazardous to maintain one code for simulation and another for real deployment, and try to sync up as progress is made. Second, while a distributed application is inherently more difficult to debug than a centralized one, we would like the users to spend their debugging energy in one address space as much as possible. Finally, when required, WiDS should support large-scale performance study for system scales approaching that of the real deployment.

Since a distributed system is essentially a collection of autonomous state machines, WiDS adopts an event-driven and object-oriented programming model, and is implemented using C++. A WiDS object represents a protocol instance or a service, and is identified by the tuple <WIDSNODE, WIDSSTUB>, analogous to how a networked service is addressed in the real world. WiDS objects exchange asynchronous messages to each other. Each message is dispatched to the target object's corresponding handler, which was declared using a macro. WiDS also provides periodic and one-time timers so that users can implement various failure detection mechanisms.

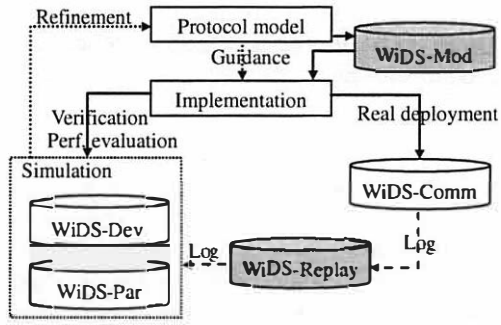


Figure 1. The ecosystem of WiDS-based protocol development and its five major components. The shaded ones (WiDS-Mod and WiDS-Replay) are under development.

These APIs isolate a WiDS-programmed protocol from any particular runtime that users want to employ. The WiDS runtimes fall into two general categories. The first is the *simulation mode*, where the runtime inserts and dispatches events through event wheel(s). Simulation mode supports pluggable topology models, allowing users to exercise different code paths in the protocol. The timestamp of a message is the source object's virtual clock plus the delay specified by the topology model. Eventwheel(s) ensure the chronological order of message execution, which in turn advances the simulation time. The simulation can be run on a single machine (linked with WiDS-Dev), enabling debugging of multiple instances of a protocol in the same address space. Alternatively, the simulation can be run in parallel on a cluster of machines to investigate performance issues for very large scales (linked with WiDS-Par). In the *network execution mode*, WiDS provides a socket-based library (WiDS-Comm), yielding a system ready to run in the real network environment. WiDS users always work with the same code base, invoking different runtimes by simply re-linking to different libraries according to their needs. Figure 1 summarizes these components of the WiDS development lifecycle. Two new members of the WiDS package, WiDS-Mod and WiDS-Replay, will be introduced in Section 4.

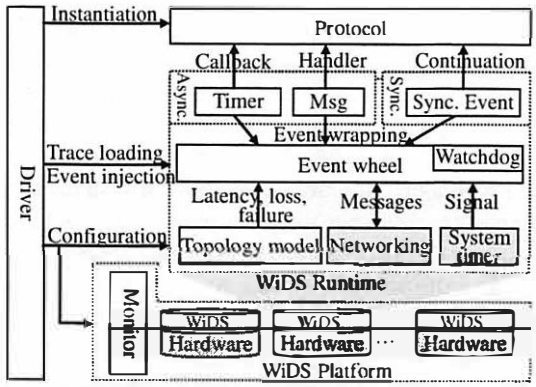


Figure 2. The WiDS architecture. Different runtimes are shaped by integrating some of the four modules: topology model, networking, system timer and event wheel.

Figure 2 depicts the WiDS runtimes. It contains topology models that generate latency and state for links between two simulated nodes, a *crystal* to trigger physical time signals, networking support based on native sockets to transport messages across physical machine boundaries, and an event wheel that stores all the events encapsulating messages, timers, and synchronous calls. Different WiDS runtimes are shaped by integrating some of these functionalities and, more importantly, different scheduling mechanism in the event wheel. There is a watchdog facility to check the progress of events, which is especially important to deal with stragglers in large-scale simulation. The monitor offers interactive simulation ability so that the user can break or step at event granularity. Along with the protocol, the user must also supply a driver program to instantiate the protocol instances, feed inputs, and inject events. In the simulation mode, the driver also specifies the topology model and node behavior (e.g., crash or create).

The WiDS parallel simulation is master-slave architected and proceeds in rounds. During each round, the master calculates a safe window (of simulation time) by looking at the head events of the slaves, and then informs the slaves to execute any events within that window. This barrier model becomes increasingly inefficient with more machines. To improve simulation performance, we have developed an optimization called Slow Message Relaxation (SMR) that simulates a window of ticks per round. This raises the possibility that a

slave machine has already advanced its simulation clock when an event with a smaller timestamp arrives. We call such a message a *Slow Message,* and simply set its timestamp to the node's current clock value before passing it to its handler. The rationale is that this is as if the message had suffered some extra delay in the network. A correctly designed distributed protocol should have already handled any network-jitter generated abnormality. However, slow messages may lead to problems that otherwise would not have appeared such as false timeouts, and may change the statistics that the simulation is measuring as well. Our analysis shows that as long as the window width is kept under some value (automatically derived from the timer APIs), there will be negligible impact. Furthermore, the window width can be adaptively adjusted to achieve the optimal performance at runtime. We have verified that this optimization achieves an order of magnitude performance improvement simulating several large scale P2P protocols, without compromising the statistical accuracy of the simulation results [6].

3. Experience with WiDS

3.1 Complete system development

One of the complete distributed systems we have developed is the BitVault data retention platform [4]. BitVault employs commodity PCs as building blocks to construct a distributed backup service that is scalable, highly reliable, and highly available. Topology-wise, nodes are arranged in a ring. At the bottom layer, there is a voting-based failure detector to monitor the health of each node by a constant number of its neighbors. A failure or new node join event is then broadcasted in O(log*N*) steps to all other members, and anti-entropy is employed to ensure the eventual convergence of membership. These protocols comprise an eventual consistent membership protocol. Above that, a placement policy places multiple replicas on a constant number of nodes, and a distributed indexing mechanism tracks the location of an object. We use massively parallel repair to deliver order-of-minutes repair time for a failed disk upon the notification of membership change. There is a scalable monitoring infrastructure embedded in the system to trigger load balancing automatically. BitVault is entirely developed and maintained using WiDS. Each BitVault node comprises several objects that implement different functions (e.g., membership, monitoring, index, data etc.), and these objects communicate with each other using WiDS messages. BitVault is robust enough that we plan to roll out a 32-node installation as an interactive backup service in the first half of this year.

Although WiDS significantly improved the development process of BitVault, during the course we have learned several important lessons that lead to the further development and research focuses for WiDS. First, while the event-based programming model is a natural fit to implement state machines, it is still difficult to program and debug. This is especially true for protocols that have multiple phases. For those protocols, the event model will spread the protocol logic in multiple event handlers, and the program must therefore explicitly handle the context moving from one handler to the other. A protocol that is multi-phased but deals with a single remote party is most easily programmed using a single thread with remote procedure calls (RPC). However, the thread model falls short if the protocol has a concurrent phase that involves multiple parties, since it must spawn separate threads to deal with these parties and then sync-up later on. The thread model must also carefully guard critical sections, which is non-trivial and something that the event model does not need to handle. Many distributed protocols, however, are in fact both multi-phase and multi-partied (e.g., two-phase commit). A good number of BitVault protocols fall into this category. Therefore, in terms of programming effort, neither the event nor the thread model is an ideal fit. These experiences motivate us to develop both new APIs and architecture to further mitigate the program burden (c.f. Section-4.1).

Second, the WiDS runtime schedules at event granularity. This implies that events are handled in turn, and one's execution can not be preempted by others. It is usually not a problem. However, consider an event that is sandwiched by two heartbeat events. If the middle event takes an exceptionally long time to complete (e.g., a blocking disk I/O) then the timer logic can be violated. In the case of BitVault, it is possible for the failure detector to wrongly signal the crash of a node, allowing the repair mechanism to kick in, which can only make things worse. This particular issue can be resolved by offering a failure detection service inside the WiDS runtime so that one can register the interested endpoints and be notified when an endpoint fails to respond. By decoupling the dependency, the probe and response can run in parallel with the execution of events, fulfilled by the WiDS runtime. However, at its core, the issue is the handling of time-critical events and the provisioning of some level of real-time guarantee. Since objects typically implement a service (e.g., the membership protocol), and the WiDS objects communicate only through messages, one thing we plan to do is to allow events of more time-critical objects to preempt other events. The other possibility is to develop a `Yield` API so that the user can chop a long-running event.

Third, related to the above two issues, many of the bugs did not manifest until the system was run in network execution mode, no matter how hard we tried to stress

the code path in simulation mode. One reason is that event handling can take arbitrarily long in network execution mode, as opposed to one (simulated) clock tick in simulation. Thus the sequence of events can differ in unexpected ways, making it difficult to discover those bugs in the simulation environment. This experience propels us to develop WiDS-Replay (Section 4.2), which logs events and deterministically replays them in simulation mode. That is to say, we'd like to build a two-way street between WiDS-Dev and WiDS-Comm.

3.2 Large-scale testing

PNRP [5] is a P2P name resolution protocol with a target scale of tens of millions of nodes. Working with our product division partners, we ported PNRP to run on WiDS, and used WiDS-par to understand its macro-behavior. We have successfully completed many simulation runs of more than a million PNRP instances using hundreds of PCs. Some of the simulations took weeks to complete. This work has allowed us to gain insights into the system behavior, identify performance and network overhead, and remove design limitations that become apparent only under stress and at such a large scale.

Running a large-scale program on a cluster of machines almost inevitably brings up the same set of (mundane) issues. These include deploying and version-controlling the code, monitoring the health of the runs, managing the cluster, dealing with stragglers, and gathering statistics for final analysis. Moreover, heterogeneity in both software and hardware is much more than a performance (and hence configuration) issue. We ran into cases where some machines were equipped with mobile NICs or had stale network drivers and therefore could not handle bulk traffic. In both cases we ran micro-benchmark with a binary search strategy to isolate them. Clearly this process needs to be automated. Finally, we also realized that the master-slave architecture of WiDS-par needs to be changed if we are to attempt scales beyond a few million protocol instances.

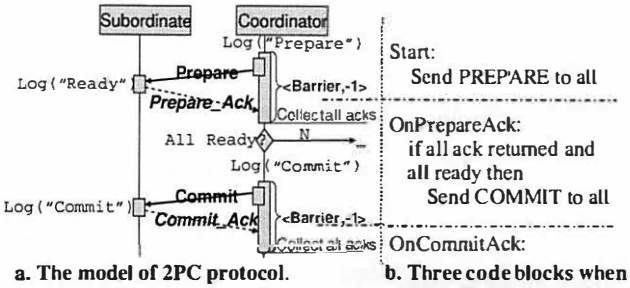
Another approach we are considering is to swap states to disk and use intelligent prefetching policies to overlap the time of loading state from disk with simulation computation. By boosting per-machine simulation scale, we hope to reduce the number of total machines needed and thus the barrier overhead.

4. Research in Progress

4.1 WiDS-Mod

A typical development process starts with some pseudo-code that bridges protocol logic with the real implementation. Currently WiDS covers the development process starting from the implementation stage. The problem is that there is a large gap between the protocol logic and the final codes, resulting in coding as well as maintenance difficulties. This is especially problematic when there are many complicated and intertwined protocols involved in a system (as in BitVault).

WiDS-Mod borrows the principle of Intentional Programming [7] and adopts a hybrid approach. Taking advantage of temporal logic [8] and UML [9], our description language allows users to specify protocol logic in an abstract level and in the GUI (e.g. Figure 3(a)). The protocol logic is then automatically turned into skeleton code (c.f. Figure 3(c)). The users then fill in the rest of implementation, such as the code that examines the field of the `AckBuf` returned from the slaves to set the `all_ready` flag that decides whether to proceed to the commit phase of a two-phase commit protocol.



a. The model of 2PC protocol.

b. Three code blocks when writing 2PC protocol in a pure event-driven fashion.

c. Sample code generated from the model in a.

```
LOG("Prepare");
PAR_BEGIN(-1)
  for_each(p, SubodinateSet) {
    SendMsg(p, PREPARE, AckBuf[p]);
  }
PAR_END
if (all_ready) {
  LOG("Commit");
  PAR_BEGIN(-1)
    for_each(p, SubodinateSet) {
      SendMsg(p, COMMIT, AckBuf[p]);
    }
  PAR_END
}
```

Figure 3. WiDS-Mod: (a) the model of the 2PC protocol; (b) codes using event-driven programming (coordinator side); (c) Sample code generated from the model.

This approach shrinks the gap between the high-level protocol specification and implementation, which is itself broken down into the logic level and the detailed handler level. Our point is that, for distributed system, these two levels already have inherently different natures and complexities (e.g., logical versus implementation correctness), so we might as well program them in different ways.

As we discussed in Section 3.1, many distributed protocols work in phases, each of which may involve multiple remote entities. A number of BitVault protocols fall into this category. For these protocols, a purely event-driven programming model quickly becomes awkward.

Figure 3(a) shows the classic two-phase commit protocol, and the three separate code blocks (Figure 3(b)).

Independent of the modeling effort, therefore, we need to extend both the WiDS APIs and the runtime. For instance, `SendMsg()` is a synchronous call which will block the caller until the destination has processed the message and sent back acknowledgement, and `PAR_BEGIN/PAR_END` closure offers a barrier semantic, which will parallelize all synchronous messaging operations inside and resume execution when all of them are finished. With user-level threading [10], we will be able to wrap the synchronous calls in the continuation events and offer thread-like semantics, and can additionally accommodate multi-party semantics, something that the pure thread model has difficulty to do. All these attempts are to further reduce programming difficulties while leveraging the strengths of both the event and thread model.

4.2 WiDS-Replay

By exercising different network models, a good portion of protocol bugs can be rooted out. Unfortunately the remaining bugs, which will only surface in the network execution mode, are also the more difficult ones to find. In comparison, the cyclic debugging process [11] we are so used to in analyzing bugs in sequential applications, in which one sets a debugging point and repeats the execution, quickly becomes too much to afford. And yet writing out and then analyzing logs is also a grueling exercise. WiDS-Replay is a set of utilities aimed at analyzing bugs that occur only during the network execution by bridging with the simulation mode.

The general methodology of WiDS-Replay is straightforward. When running in the network execution mode, checkpoints are executed at each machine for all important states, and logs are also kept for any inputs between the checkpoints that may change the state of a running protocol instance (file I/O, wall-clock, random number generators, etc.). Finally, user-defined logs are coalesced and dumped into the same log file. We then start the protocol in the simulation mode, reloading the checkpoint and log traces to reconstruct context. Notice that in the network mode every instance is running as a separate process, whereas in the simulation mode each instance is a WiDS object. Therefore we carefully perform data marshalling and de-marshaling to make sure that the object states are loaded correctly. In the replay phase, we navigate the traces at the granularity of log entries while bringing up the code alongside as the navigation context. We then use deterministic forward and backward replay to examine the program state, doing this across different objects (and hence protocol instances running on different machines) when necessary.

The object-oriented programming model of WiDS makes it possible to replay a distributed protocol within a single address space and on a single machine. Therefore, WiDS-Replay provides the capability of *virtualizing* the distributed system debugging process. A prototype of WiDS-Replay has already been built, but much more needs to be researched and developed before it can be put into practice.

WiDS-Replay can also work within the simulation mode. Here, periodic checkpoint is sufficient for deterministic replay, assuming that the simulation environment is also checkpointed. One may argue that since simulation is deterministic anyways, why bother with checkpointing. The truth is that for a complex protocol, it often takes a long time to reach a faulty point. Checkpointing segments the debugging process and allows the user to invoke different debugging details when appropriate.

5. Related Work

As observed in [12], sharing the same code base for the purpose of development, simulation, and deployment is a popular notion. There have been some attempts along the same line. For instance, Neko [13] is a java platform that allows the same algorithm to run both in simulation and in real network. Though we do share the same philosophy, their interfaces and architecture are quite different from ours. Neko does not offer parallel simulation capability, and it is not clear whether it has been used to build a complete system. While WiDS offers native C/C++ support, MACEDON [14] takes a different approach by offering a domain-specific language for FSM (finite state machine) based protocols. The MACEDON approach is geared towards quick prototyping overlay applications. Large-scale performance study requires an emulation approach (discussed below), though it should be possible to add PDES (Parallel Discrete Event Simulation [15]) support as well. One thing that MACEDON does very well is to abstract many common services of overlay systems into generic packages. WiDS can take the same approach for services such as failure detector and membership protocols, which are common building blocks for distributed system.

One contribution of the current WiDS package is its capability of performing large-scale simulation and testing on clustered machines. While there have been many works on PDES, our Slow Message Relaxation optimization is unique in that it takes advantages of the time slacks that all distributed protocols use to cope with

unreliable network transmission. A related approach to large scale testing is *emulation*, which is exactly the same as the network execution mode of WiDS except that many (typically thousands of) instances of protocols run on each testing node, and the packets are routed through a cluster of machines modeling the Internet topology and (therefore) packet delays [16][17]. There are pros and cons in these two approaches, and it will be an interesting research topic to identify synergy.

The versatility of WiDS extends to cover other important aspects of the development process. WiDS-Mod borrows principles from Intentional Programming [6] to abstract high level logic (intention) from implementation. WiDS-Mod provides a natural and formal model, and yet reserves sufficient flexibility for developers to describe their implementation details.

The idea of using checkpoint and logging at runtime to discover difficult bugs using deterministic replay is an old one [18]. WiDS-Replay checkpoints and logs distributed protocols as they are run in the real environment, but deterministically replays and debugs the protocols on a single machine within one address space. As far as we know, this is a novel approach.

6. Conclusion

WiDS was born in response to many early lessons we learned when researching and developing several P2P protocols. As an integrated toolkit that covers rapid prototyping, large-scale simulation, and deployment, it has already significantly improved our productivity. Still, to become truly holistic, WiDS must evolve further to address the difficulties of programming as well as debugging distributed systems.

Acknowledgement

We would like to thank Noah Horton, Geogy Samuel, Brian Lieuallen and Sandeep Singhal for the support of running large-scale simulation of the PNRP protocols using WiDS. We also thank the anonymous reviewers and Richard Draves and Kurt Akeley for their valuable inputs.

Reference:

[1] P. Druschel and A. Rowstron, "PAST: A large-scale, persistent peer-to-peer storage utility", in Proc. HotOS VIII, Schloss Elmau. Germany, May 2001

[2] John Kubiatowicz, David Bindel etc., "OceanStore: An Architecture for Global-Scale Persistent Storage", in Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000), November 2000

[3] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. "Ivy: A Read/Write Peer-to-peer File System", in OSDI, December 2002

[4] Zheng Zhang, Qiao Lian, Shiding Lin, Wei Chen, Yu Chen, Chao Jin, "BitVault: a Highly Reliable Distributed Data Retention Platform", under submission

[5] Microsoft TechNet, "Introduction to Windows Peer-to-Peer Networking", http://www.microsoft.com/technet/prodtechnol/winxppro/deploy/p2pintro.mspx

[6] Shiding Lin, Aimin Pan, Rui Guo, Zheng Zhang, "Simulating Large-Scale P2P Systems with the WiDS Toolkit", to appear in 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication (MASCOTS 2005)

[7] C. Simonyi. "The Death of Computer Languages, The Birth of Intentional Programming", Technical Report MSR-TR-95-52, Microsoft Research, 1995

[8] Leslie Lamport, "Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers". Addision-Wesley (2002).

[9] Unified Modeling Language 1.5, OMG, http://www.omg.org/technology/documents/formal/uml.htm

[10] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. "Capriccio: Scalable Threads for Internet Services", In Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP), 2003.

[11] Joel Huselius. "Debugging parallel systems: A state of the art report". Technical Report 63, Mlardalen University, Department of Computer Science and Engineering, September 2002

[12] Michael Jones and John Dunagan, "Engineering Realities of Building a Working Peer-to-Peer System", MSR Technical Report MSR-TR-2004-54. June 2004.

[13] P. Urban, X. Defago, and A. Schiper, "Neko: A single environment to simulate and prototype distributed algorithms", in Proc. of the 15th Int'l Conf. on Information Networking (ICOIN-15), (Beppu City, Japan), Feb. 2001.

[14] A. Rodriguez, C. Killian, S. Bhat, D. Kostic, and A. Vahdat, "MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks". in Proc. of the USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI), 2004.

[15] A. Ferscha, and S.K. Tripathi, "Parallel and distributed simulation of discrete event systems". Technical report, University of Maryland, August 1994.

[16] Amin Vahdat, Ken Yocum, Kevin Walsh, etc., "Scalability and Accuracy in a Large-Scale Network Emulator", in OSDI, December 2002

[17] Emulab project, the Utah network testbed (Web site). http://www.emulab.net/.

[18] E. N. M. Elnozahy, L. Alvisi, Y. Wang, and D. B. Johnson. "A survey of rollback-recovery protocols in message-passing systems", ACM Computing Surveys (CSUR), 34(3):375-408, 2002

Causeway: Operating System Support For Controlling And Analyzing The Execution Of Distributed Programs

Anupam Chanda, Khaled Elmeleegy, and Alan L. Cox
Department of Computer Science
Rice University, Houston, Texas 77005, USA
{anupamc,kdiaa,alc}@cs.rice.edu

Willy Zwaenepoel
School of Computer and Communication Sciences
EPFL, Lausanne, Switzerland
willy.zwaenepoel@epfl.ch

Abstract

In this paper we introduce Causeway, operating system support facilitating the development of meta-applications, like priority scheduling and performance debugging, that control and analyze the execution of distributed programs. Meta-applications use Causeway to inject and access metadata on application execution paths to implement their specific goals. Causeway has two components: (1) interfaces to inject and access metadata and (2) mechanisms to automate propagation of metadata. Using Causeway we could rapidly implement a distributed priority scheduling system where priority of a task is injected and propagated as metadata, and accessed to implement global priority scheduling. This required writing only about 150 lines of code on top of Causeway. With this system we demonstrate global priority scheduling on an implementation of the TPC-W benchmark.

1 Introduction

In this paper we introduce Causeway, operating system support facilitating the development of *meta-applications* that control and analyze the execution of distributed programs. Priority scheduling and performance debugging are examples of such meta-applications. A meta-application can span across the application and the operating system (kernel and libraries). Meta-applications use Causeway to inject and access *metadata* on application execution paths to implement their specific goals, e.g., scheduling or debugging. Causeway performs automatic propagation of injected metadata along application execution paths enabling the meta-application to access metadata from anywhere along those paths.

Causeway has two components: (1) interfaces for *actors* to inject and access metadata and (2) mechanisms to automate propagation of metadata to and from actors across *channels*. An actor is an execution context; it can be a process, a thread (in a multithreaded program) or an event handler (in an event-driven program), whether executing in user or kernel mode. Application execution is performed by one or more actors. An actor may communicate with other actors during an execution. A channel is defined as the means of communication between two (or more) actors. Metadata is arbitrary data that is distinct from application data but is propagated alongside application data through the execution paths of the distributed program. Causeway interfaces can be called from both the application and the operating system. Causeway automatically propagates metadata between actors across channels without the need for any application modification.

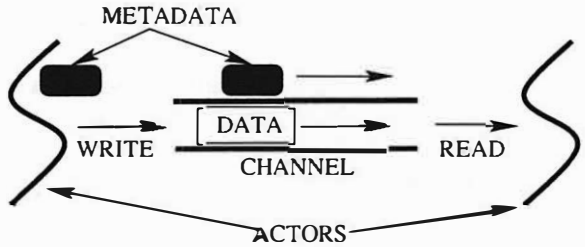


Figure 1: Propagation of Metadata Between Two Actors Across a Channel

At an abstract level, Causeway works as follows. Metadata is associated with an actor when that actor performs injection. Later, when the actor writes application data to a channel, its metadata is associated with the application data written. On a subsequent read from the channel by either the same or a different actor, the metadata is propagated to the actor performing the read. Figure 1 illustrates the concept of propagation of metadata between two actors across a channel.

The complete set of channel types are: (1) sockets, (2) pipes, (3) files, and (4) shared memory. Causeway propagates metadata along a channel on read and write operations by an actor. Some of these channel types are visible to the operating system (kernel and libraries) while oth-

ers are not. Pipes, sockets and files are system visible whereas shared memory is not. Further, some channel types are persistent, e.g., files, while others, like shared memory, are short-lived. Causeway currently propagates metadata across socket and pipe channels. As ongoing work we are adding support in Causeway for file and shared memory channels.

There are quite a few challenges in the design of Causeway. First, when metadata is propagated to an actor, a decision needs to be made about what to do with the existing metadata on the actor. It is possible that the incoming metadata pertains to a new request to the system: in this case the incoming metadata needs to be assigned to the actor which loses its existing metadata. Alternatively, the incoming metadata may be associated with the same request as being currently executed by the actor but carry a different value: in this case some composition of the incoming metadata and the existing metadata needs to be applied to the actor. Second, on a read on a channel, different pieces of data may be associated with different metadata. Again, a decision is required about what metadata to propagate to the actor. Finally, handling channels invisible to the system, e.g., shared memory, is a challenge in itself. We address these issues in Sections 4 and 6.

We have implemented Causeway in the FreeBSD operating system kernel, the `libpthread` and the `libevent` [8] libraries. Causeway, thus, achieves automatic propagation of metadata without the need for application modification.

Using Causeway we could rapidly implement a distributed priority scheduling system where priority of a task is injected and propagated as metadata, and accessed to implement global priority scheduling. This required writing only about 150 lines of code on top of Causeway. With this system we demonstrate global priority scheduling on an implementation of the TPC-W [10] benchmark used as a test distributed program. This distributed program includes a Web server, an application server and a database, all running on different machines. Each request for service is assigned a priority. This priority is then passed as metadata which follows *all* actors performing the execution for this request in the Web server, application server and the database. No modification of the TPC-W benchmark, other than selective injection of priority, was required.

Causeway is not the first system to advocate the propagation of metadata along request execution paths in distributed systems. Earlier work in Domain and Type Enforcement (DTE) in Unix systems [2] and Stateful Distributed Interposition (SDI) [9] employ metadata or context propagating mechanisms similar to Causeway. While DTE propagates the type of data written by a sending process and the domain of the sending process for interprocess communication to implement security policies, Causeway extends this mechanism to propagate arbitrary types of metadata across different kinds of channels for a variety of meta-applications. The work closest to Causeway is SDI [9] which also provides metadata or context propagating mechanism for multitiered servers. Causeway differs from SDI in two aspects: first, Causeway propagates the value of the metadata across channels and not its reference as in SDI, and, second, we want to extend Causeway to handle shared memory channels. Shared memory channels occur frequently in many programs, e.g., Apache and MySQL which are used extensively to build distributed applications.

Several examples of meta-applications appear in literature; they have generally been built from scratch. Aguilera et al. [1] infer causal paths from message traces to locate nodes causing performance bottlenecks. The use of request tagging has been utilized to determine faults in Internet services [4]. The resulting Pinpoint system uses instrumentation of the J2EE platform to pass on request identifiers among the different components of the system. These meta-applications, and many more, can be implemented on top of Causeway.

Magpie [3, 5] represents a different approach to the analysis of distributed programs. Magpie logs events, and extracts events belonging to a particular request execution by performing temporal joins over this log. These joins are based on application-specific schemas, which may require considerable expertise and knowledge about the application. Magpie can measure per-request resource utilization in a distributed program. Magpie and request identification using Causeway present an interesting set of tradeoffs. Magpie does not require kernel or library modifications, and leverages event logging facilities already present in Windows. In contrast, Causeway accepts the premise of such modifications, and as a result avoids the need for detailed knowledge about the application.

Traditionally, there have been two approaches to writing such meta-applications: a log-based approach and a metadata-passing approach. In a log-based approach, a log is maintained to record events triggered as requests are executed. Logs on the different components of the system are later merged and analyzed. Magpie utilizes this approach. Metadata-passing approach propagates metadata along the request execution paths of a system; the propagated metadata is accessed by the meta-application. For example, Pinpoint passes request identifiers along request execution paths and utilizes them to identify faulty components in the system. A meta-application using the metadata-passing approach can affect the execution of requests in an online manner; however, a log-based approach cannot achieve the same because although collection of log is online, its processing

lags the execution of requests by a positive delay. Causeway adopts the metadata-passing approach and provides operating system support for the common aspects of meta-applications that can be built using this approach.

With Causeway users can implement tasks like priority scheduling and performance debugging of distributed programs. Such users are different from the class of operating systems developers and application developers. Meta-application developers use the interfaces exported by Causeway to implement the desired meta-application requiring little knowledge of the application or the operating system. By separating development of meta-applications from applications, Causeway parallels the concept of Aspect-Oriented Programming [7] which allows developers to dynamically modify static application to achieve secondary goals without modifying the original static model.

The rest of this paper is organized as follows. We justify the need for a framework like Causeway in Section 2. We give a detailed specification of metadata in Section 3. Section 4 presents an overview of Causeway's design. We give demonstration of Causeway's use in Section 5. Ongoing and future work is outlined in Section 6. We conclude in Section 7.

2 Need for a Framework

In this section we motivate why the operating system should support metadata injection, access, and propagation. In other words, we answer the question — "why not build the support into applications".

First, we note that the use of metadata is significantly different than the (application) data. Hence, from a software engineering viewpoint, there is a logical separation between how data and metadata are handled.

Second, propagating metadata at application level only will involve augmenting applications and application-level inter-process communication protocols. This approach has its own pitfalls. Consider a multi-tiered server for web services. Let us assume, an application-specific HTTP header is defined to propagate metadata to a web server. But not all applications use the same protocol. For instance, the web server may need to communicate to a database server. In this case, the database server does not understand HTTP. To propagate metadata to the database server, then, the communication protocol between the web server and the database server needs to be augmented as well. In essence, by this approach all possible application-level communication protocols will require augmentation — a tedious solution. By making the propagation of metadata a system-level function, it becomes independent of the application-level communication protocol being used.

Finally, in a distributed program, it is possible that some individual components are unaware about the presence of metadata or ignore it. Consider a 3-tier system, where the middle tier application is unaware of metadata. The front and the back-end tiers may still, however, need to access metadata. In this scenario, operating system support for automatic metadata propagation is required in the middle tier even though the middle tier application may remain ignorant to metadata.

One may implement this framework support into middlewares. This approach will work when all components of the application are built using such middlewares. However, this approach is not sufficient for all cases and kernel modifications may be required. For example, our implementation of the TPC-W benchmark consists of the Apache (version 1) web server. Apache is a process-based web server, and thus the distributed priority scheduling system may need to change the priorities of Apache processes. This may only be attained by kernel modifications. Hence, we argue that kernel modifications are a necessary feature of the framework support for meta-applications.

3 Metadata

Metadata in Causeway is a five-tuple of *(identifier, type, value, propagation bit mask, merge routine identifier)*. On injection, a metadata object is created and assigned an immutable, system-wide unique identifier. Type and value are self-explanatory. Meta-applications can define new metadata types, if required. The propagation bit mask contains a flag per channel type signifying whether this metadata object is propagated across channels of that type or not. The merge routine identifier specifies which *merge routine* should be invoked, when required. A merge routine takes two or more metadata objects as input and combines them to produce a single metadata object. Causeway implements frequently used merge routines like `min`, `max`, `concat`, etc. Other merge routines can be implemented in Causeway, if required. A merge routine is invoked on the incoming metadata and the existing metadata of an actor when they have the same identifier but differ in value.

4 Causeway Design

Causeway has two components: (1) interfaces to inject and access metadata and (2) mechanisms to automate propagation of metadata.

4.1 Interfaces

Meta-applications can interact with Causeway in two ways — through an interface by which actors can inject and access metadata, and through a callback interface under which Causeway calls handlers registered by the meta-application.

Actor Interface Causeway provides interfaces for injection, inspection, modification and removal of metadata by actors. These interfaces may be called from user-level or kernel-level by an actor, which could be a process, a thread or an event-handler.

Causeway defines the following interface functions to be called by an actor: `cwa_type_query` retrieves the collection of metadata types that are associated with the actor; `cwa_data_lookup` retrieves any metadata of the given type that is associated with the actor; `cwa_data_insert` associates the given metadata with the actor, overwriting any prior metadata of that type; and `cwa_data_remove` disassociates any metadata of the given type from the actor. Since all metadata are actor-private, synchronization of metadata access interfaces is not required.

Callback Interface Using Causeway's callback interface the meta-application can register a *transfer-point* callback method. A transfer point is a point where data is read from or written to a channel by an actor. At a transfer point Causeway determines if the type of the metadata being passed has a callback method registered. If a callback method exists, it is invoked with the metadata as its argument. The callback method reads and possibly modifies the metadata and passes it back to the transfer point. The callback method can call arbitrary operating system code, e.g., to change the priorities of actors.

4.2 Automatic Propagation of Metadata

When an actor performs a write on a channel, the actor's metadata is associated with the data written into the channel. On a subsequent read on the channel by an actor, metadata is propagated from the data and assigned to the actor. First, we describe the rules of metadata assignment to an actor. Then we describe the propagation mechanism across each of the channel types.

4.2.1 Assigning Metadata to an Actor

There are two ways metadata can be assigned to an actor - injection and propagation across a channel. On injection, an actor loses any existing metadata and the injected metadata is assigned to it. On propagation, two cases are possible. First, the actor does not have any existing metadata, or the identifier of its existing metadata does not match the identifier of the metadata propagated. In this case the actor loses its existing metadata, if any, and the propagated metadata is assigned to it. Second, the identifier of the actor's existing metadata matches that of the propagated one but the metadata values are different (no action is required if the values match). In this case the merge routine, specified in the metadata, is invoked on the two metadata, and the result is assigned to the actor.

4.2.2 Propagation across Channels

Now we describe the propagation mechanism across each of the channel types. We emphasize that the rules described in Section 4.2.1 are applied to assign metadata to an actor after propagation across a channel. Causeway currently implements metadata propagation across sockets and pipes.

Sockets and Pipes Causeway handles sockets and pipes similarly. When an actor writes to a socket (or a pipe), Causeway associates metadata from the actor to the data written. On subsequent read from the socket by another (or the same) actor, metadata is propagated from the data to the actor.

The above applies for `LOCAL` sockets only. For `INTERNET` sockets, data is encapsulated in IP packets for send and receive across sockets. Causeway encapsulates metadata, in addition to data, in the IP packets. For IPv4, Causeway encapsulates metadata in the IP header as IP options. In particular, Causeway defines a new IP option type, populates the IP header with the option type, option length, and option payload. At the receiving side, the metadata, if any, is extracted from the IP options. Since IP options can be a maximum of 40 bytes only, with 1 byte each for options type and options length, Causeway can transfer at most 38 bytes of metadata via this mechanism. For most practical purposes, this has proven sufficient. This limitation is an artifact of Causeway's implementation and not its design. A general purpose tunneling protocol could be used to overcome this limitation, if required. For IPv6, Causeway uses the destination options in the IP header which does not have any size limitation. Further details about that are outside the scope of this paper.

The following case presents a challenge to the above design. Consider a scenario where multiple pieces of data are ready to be read from a socket (or pipe), and at least one piece has a metadata identifier different than rest of the above. Then a decision needs to be made about what metadata is to be propagated to the actor reading from the socket (or pipe). Causeway resolves this situation as follows. The pieces of data ready on the socket are read in a `FIFO` manner. Causeway returns from the read just before the first piece having metadata identifier different than the earlier pieces. So, all the pieces of data read by the actor are guaranteed to have the same metadata identifier. The merge routine is then applied on these metadata, if their values differ, and the result is propagated to the actor. In our implementation of Causeway on FreeBSD, we associate metadata with `mbufs` on send and receive operations on `sockets`.

5 Using Causeway

Meta-applications to control and analyze the execution of distributed programs can be built easily using Causeway. We illustrate two such meta-applications here: a multi-tier priority scheduling system and a distributed profiler.

5.1 Multi-tier Priority Scheduling System

Using Causeway we could rapidly implement a multi-tier priority scheduling system, controlling the order in which requests sent to a multi-tiered, web-based application are executed. Under this system, the application injects priority as metadata, Causeway automatically propagates the priority metadata to all the tiers, and the meta-application uses the priority metadata to enforce priority scheduling on each tier. The meta-application is automatically invoked on each tier through Causeway's callback mechanism.

The implementation of this system on top of Causeway required writing only about 150 lines of code. We tested this system with an implementation of the TPC-W benchmark [10]. No modifications were made to the TPC-W code, other than selective injection of priority. We subjected the TPC-W system to a background workload and a foreground test load. The background workload was injected with metadata signifying default priority. The foreground load was injected with metadata for default priority in one case, and high priority for another. Response time measurement for the foreground load showed one to two orders of magnitude of improvement when using high priority.

5.2 Distributed Profiler

In this section we present the design for a distributed profiler that we are developing using Causeway. A distributed application has multiple components executing in different processes. Furthermore, these different processes may be executing on multiple machines. While it is possible to profile the components in isolation, it is hard to collate the profile information for different components to form a single, global profile. We intend to achieve this with a distributed profiler as follows. We will pass context information as metadata on remote procedure calls (RPC) from the caller to the callee. This propagated context information will be used to annotate the callee's profile information. Profile information from the caller and the callee can then be stitched together with this context information. Thus using Causeway, a single, global profile for a distributed program can be generated.

6 Ongoing and Future Work

In this section we describe the design of Causeway to propagate metadata across file and shared memory channels. As ongoing work, this design is being implemented in Causeway. As future work, we intend to extend the design of Causeway to handle parallel computation paths and address security concerns. Finally, we intend to quantify the overhead of using Causeway.

6.1 Files

When an actor writes to a file, Causeway assigns the metadata from the actor to the range of bytes written. On a read operation, two cases are possible: (1) all bytes read are associated with the same metadata — the metadata is propagated to the actor in this case, (2) at least one byte has associated metadata different than the rest — in this case the merge routine, specified in the metadata, is applied on the different metadata, and the result is propagated to the actor.

6.2 Shared Memory

Producer-consumer is a popular model of shared memory usage. This model is used, by applications like Apache and MySQL. At an abstract level, the model works as follows. Producers and consumers share a buffer or queue of objects. A producer creates an object, acquires a lock to enter the critical section, adds the object to the shared buffer or queue, and releases the lock. A consumer acquires a lock to enter the critical section, retrieves and removes an object from the shared buffer or queue, releases the lock, and then accesses the retrieved object. The use of system-supported synchronization primitives, like `pthread_mutex` or `pthread_rwlock`, can make producer-consumer communication through shared memory visible to Causeway.

We note that the producer accesses the created object just *before* the *lock* operation and in the critical section, while the consumer accesses the retrieved object in the critical section and just *after* the *unlock* operation. We are investigating ways to identify this pattern and insert (in the source or precompiled binary) calls to save metadata from the producer and calls to retrieve metadata in the consumer. The transformed producer code will do the following: create the object, save the producer's metadata and associate it with the created object; then enter the critical section as in the unmodified program. After the critical section, the transformed consumer code will do the following: access the retrieved object and retrieve the metadata associated with the retrieved object.

6.3 Execution Path Fork and Join

Causeway needs to handle execution path *fork*s and *join*s caused by parallel computation paths. In the common case, an actor writes to a channel and then reads from the same channel, waiting for a response. However, sometimes, an actor may write to multiple channels without waiting for the individual responses. As an example, a web server may send queries to multiple nodes in a repli-

cated database system and then wait for their individual responses. Each of these writes constitutes a fork in the execution path. When the response corresponding to a fork arrives, it is termed a join. In the above example, the response from a database server constitutes a join. As future work, we intend to extend the design of Causeway to identify and handle such forks and joins in the execution paths.

6.4 Security Concerns

Like SDI [9] we argue that the issue of illegal network access modifying metadata in IP packets should be addressed by using IPSec [6]. In order to prevent the illegal modification of the metadata by the application, we intend to incorporate a secure signing mechanism like MD5 as a part of the metadata for propagation across the user-kernel boundary.

7 Conclusions

The contributions of this paper are the following. We have designed Causeway, operating system support for facilitating development of meta-applications, like priority scheduling and performance debugging, to control and analyze the execution of distributed programs. Causeway provides interfaces for metadata injection and access, and performs automatic propagation of metadata in distributed programs. Propagated metadata can be accessed and used to implement the desired service in the system. We have implemented Causeway in the FreeBSD operating system, the `libpthread` and the `libevent` libraries. We have demonstrated the use of Causeway by implementing a multi-tier priority scheduling system and using it to achieve global priority scheduling on an implementation of the TPC-W benchmark [10].

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 74–89, Oct. 2003.

[2] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. A Domain and Type Enforcement UNIX Prototype. In *Fifth USENIX UNIX Security Symposium*, June 1995.

[3] P. T. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *OSDI*, pages 259–272, Dec. 2004.

[4] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Proceedings of the International Conference on Dependable Systems and Networks (IPDS Track)*, pages 595–604, June 2002.

[5] R. Isaacs, P. Barham, J. Bulpin, R. Mortier, and D. Narayanan. Request extraction in Magpie: events, schemas and temporal joins. In *SIGOPS EW'04: ACM SIGOPS European Workshop*, Sept. 2004.

[6] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. In *IETF RFC 2401*, 1998.

[7] ONJava.com. Introduction to Aspect-Oriented Programming. At http://www.onjava.com/pub/a/onjava/2004/01/14/aop.html.

[8] N. Provos. Libevent - an event notification library. Version 0.7c is available from the author's web site, http://www.monkey.org/~provos/libevent/, Oct. 2003. Libevent is also included in recent releases of the NetBSD and OpenBSD operating systems.

[9] J. Reumann and K. G. Shin. Stateful Distributed Interposition. *ACM Transactions on Computer Systems*, 22(1):1–48, Feb. 2004.

[10] T. P. P. C. (TPC). TPC BENCHMARK W (web commerce). At http://www.tpc.org/tpcw/, Feb. 2002.

Treating Bugs As Allergies: A Safe Method for Surviving Software Failures

Feng Qin, Joseph Tucek and Yuanyuan Zhou
Department of Computer Science, University of Illinois at Urbana-Champaign
{fengqin, tucek, yyzhou}@cs.uiuc.edu

ABSTRACT

Many applications demand availability. Unfortunately, software failures greatly reduce system availability. Previous approaches for surviving software failures suffer from several limitations, including requiring application restructuring, failing to address deterministic software bugs, unsafely speculating on program execution, and requiring a long recovery time.

This paper proposes an innovative, *safe* technique, called Rx, that can quickly recover programs from many types of common software bugs, both deterministic and non-deterministic. Our idea, inspired by allergy treatment in real life, is to rollback the program to a recent checkpoint upon a software failure, and then to reexecute the program in a *modified* environment. We base this idea on the observation that many bugs are correlated with the execution environment, and therefore can be avoided by removing the "allergen" from the environment. Rx requires few to no modifications to applications and provides programmers with additional feedback for bug diagnosis.

1 Introduction

1.1 Motivation

Many applications, especially critical ones such as process control or on-line transaction monitoring, require high availability [14]. For server applications, downtime leads to lost productivity and lost business. According to a report by Gartner Group [22], the average cost of an hour of downtime for a financial company exceeds six million US dollars. With the tremendous growth of e-commerce, almost every kind of organization increasingly depends on highly available systems.

Unfortunately, software failures greatly reduce system availability. A recent study showed that software failures account for up to 40% of system failures [16]. Among them, memory-related bugs and concurrency bugs are common and severe software defects, causing more than 60% of system vulnerabilities [9]. For this reason, software companies invest enormous effort and resources on software testing and bug detection prior to releasing software. However, software failures still occur during production runs since some bugs will inevitably slip through even the strictest testing. Therefore, to achieve higher system availability, mechanisms must be devised to allow systems to survive the effects of uneliminated software bugs to the largest extent possible.

Previous work on surviving software failures can be classified into four categories. The first category encompasses various flavors of rebooting (restarting) techniques, including whole program rebooting [14, 25], micro-rebooting of partial system components [7, 6, 8], and software rejuvenation [15, 13, 4]. Since many of these techniques were originally designed to handle *hardware* failures, most of them are ill-suited for surviving software failures. For example, they cannot deal with deterministic software bugs, a major cause of software failures [10], because these bugs will still occur even after rebooting. Another important limitation of these methods is service unavailability while restarting, which can take up to several seconds [26]. For servers that buffer significant amounts of state in main memory (e.g. data caches), it requires a long period to warm up to full service capacity [5, 27]. Micro-rebooting [8] addresses this problem to some extent by only rebooting the failed components. However, it requires legacy software to be reconstructed in a loosely-coupled fashion.

The second category includes general checkpointing and recovery. The most straightforward method in this category is to checkpoint, rollback upon failures, and then reexecute either on the same machine [12, 19] or on a different machine designated as the "backup server" (either active or passive) [14, 3, 5, 27]. Similar to whole program rebooting, these techniques were also proposed to deal with hardware failures, and therefore suffer from the same limitations in addressing software failures. Progressive retry [28] is an interesting improvement over these works. It reorders messages to increase the degree of non-determinism. While this work proposes a promising direction, it limits the technique to message reordering. As a result, it cannot handle bugs unrelated to message order. For example, if a server receives a malicious request that exploits a buffer overflow bug, simply reordering messages will not solve the problem. The most aggressive approaches in this category include recovery blocks [18] and n-version programming [2, 1], both of which rely on different implementation versions upon failure. These approaches may survive deterministic bugs under the assumption that different versions fail independently. But they are too expensive to be adopted

by software companies because they double the software development costs and efforts.

The third category comprises application-specific recovery mechanisms, such as the multi-process model (MPM), exception handling, etc. Some multi-processed applications, such as the multi-processed version of the Apache HTTP server and the CVS server, can simply kill a failed process and start a new one to handle a failed request. While simple and capable of surviving certain software failures, this technique has several limitations. First, if the bug is deterministic, the new process will most likely fail again at the same place given the same request (e.g. a malicious request). Second, if a shared data structure is corrupted, simply killing the failed process and restarting a new one will not restore the shared data to a consistent state, therefore potentially causing subsequent failures in other processes. Other application-specific recovery mechanisms require software to be failure-aware, which adversely affects programming difficulty and code readability.

The fourth category includes several recent non-conventional proposals such as failure-oblivious computing [20, 21] and the reactive immune system [23]. Failure-oblivious computing proposes to deal with buffer overflows by providing *artificial* values for out-of-bound reads, while the reactive immune system returns a *speculative* error code for functions that suffer software failures (e.g. crashes). While these approaches are inspiring and may work for certain types of applications or certain types of bugs, they are *unsafe* to use for correctness-critical applications (e.g. on-line banking systems) because they "speculate" on programmers' intentions, which can lead to program misbehavior. The problem becomes even more severe and harder to detect if the speculative "fix" introduces a silent error that does not manifest itself immediately. Such problems, if they occur, are very hard for programmers to diagnose since the application's execution has been forcefully and silently perturbed by those speculative "fixes".

Besides the above individual limitations, existing work provides insufficient feedback to developers for debugging. For example, the information provided to developers may include only a core dump, several checkpoints, and an event log for the deterministic replay of a few seconds of recent execution. To save debugging effort, it is desirable if the run-time system can provide information regarding the bug type, under what conditions the bug is triggered, and how it can be avoided. Such diagnostic information can guide programmers during their debugging process and thereby enhance efficiency.

1.2 Our Contributions

In this paper, we propose a *safe* technique, called *Rx*, to quickly recover from many types of software failures caused by common software defects, both deterministic and non-deterministic. It requires few to no changes to applications' source code, and provides diagnostic information for postmortem bug analysis. Our idea is to rollback the program to a recent checkpoint when a bug is detected, *dynamically change the execution environment based on the failure symptoms*, and then reexecute the buggy code region in the new environment. If the reexecution successfully passes through the problematic region, the environmental changes are disabled to avoid imposing time and space overheads.

Our idea is inspired from real life. When a person suffers from an allergy, the most common treatment is to remove allergens from their *living environment*. For example, if patients are allergic to milk, they should remove diary products from the diet. If patients are allergic to pollen, they may install air filters to remove pollen from the air. Additionally, when removing a candidate allergen from the environment successfully treats the symptoms, it allows diagnosis of the cause of the symptoms. Obviously, such treatment cannot and also should not start before patient shows allergic symptoms since changing living environment requires special effort and may also be unhealthy.

In software, many bugs resemble allergies. That is, their manifestation can be avoided by *changing the execution environment*. According to a previous study by Chandra and Chen [10], around 56% of faults in Apache depend on execution environment[1]. Therefore, by removing the "allergen" from the execution environment, it is possible to avoid such bugs. For example, a memory corruption bug may disappear if the memory allocator delays the recycling of recently freed buffers or allocates buffers non-consecutively in isolated locations. A buffer overrun may not manifest itself if the memory allocator pads the ends of every buffer with extra space. Data races can be avoided by changing timing events such as thread-scheduling, asynchronous events, etc. Bugs that are exploited by malicious users can be avoided by dropping such requests during program reexecution. Even though dropping requests may make a few users (hopefully the malicious ones) unhappy, they do not introduce incorrect behavior to program execution like the failure-oblivious approaches do. Furthermore, given a spectrum of possible environment changes, the least intrusive changes can be tried first, reserving the most extreme one as a last resort for when all other changes have failed. Finally, the specific environment change which cures the problem gives diagnostic information as to what the bug might be.

Similar to an allergy, it is difficult and expensive to apply these execution environmental changes from the very beginning of the program execution because we do not

[1] Note that our definition of execution environment is different from theirs. In our work, the standard library calls, such as *malloc*, and system calls are also part of execution environment.

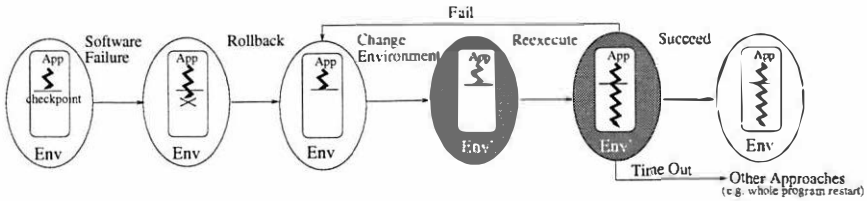


Figure 1: Rx main idea

know what bugs might occur later. For example, zero-filling newly allocated buffers imposes time overhead. Therefore, we should lazily apply environmental changes only when needed.

Compared to previous solutions, Rx has the following unique advantages:

(1) Comprehensive: Besides non-deterministic bugs, Rx can also survive deterministic bugs. We have evaluated our idea using several server applications with common software bugs and our preliminary results show that Rx can successfully survive these software bugs.

(2) Safe: Rx does not speculatively "fix" bugs at run time. Instead, it prevents bugs from manifesting themselves by changing only the program's execution environment. Therefore, it does not introduce uncertainty or misbehavior into a program's execution, which is difficult for programmers to diagnose.

(3) Noninvasive: Rx requires few to no modifications to applications' source code. Therefore, it can be easily applied to legacy software.

(4) Efficient: Because Rx requires no rebooting or warm-up, it significantly reduces system down time and provides reasonably good performance during recovery. Additionally, Rx imposes only the minimal overhead of lightweight checkpointing during normal execution.

(5) Informative: Rx does not hide software bugs. Instead, bugs are still exposed. Furthermore, besides the usual bug report package (e.g. core dumps, checkpoints and event logs), Rx provides programmers with additional diagnostic information for postmortem analysis, including what conditions triggered the bug and which environmental changes can and cannot avoid the bug. Based on such information, programmers can more efficiently find the root cause of the bug. For example, if Rx successfully avoids a bug by padding newly allocated buffers, the bug is likely to be a buffer overflow. Similarly, if Rx avoids a bug by delaying the recycling of freed buffers, the bug is likely to be caused by double free or dangling pointers.

2 Main Idea of Rx

The main idea of Rx is to reexecute a failed code region in a new environment that has been modified based on the failure symptoms. If the bug's "allergen" is removed from the new environment, the bug will not occur during reexecution, and so the program will survive this software failure without rebooting the whole program. After the reexecution safely passes through the problematic code region, the environmental changes are disabled to reduce time and space overhead.

Figure 1 shows the process by which Rx survives software failures. Rx periodically takes light-weight checkpoints that are specially designed to survive software failures instead of hardware failures or OS crashes [24]. When a bug is detected, either by an exception or by the integrated dynamic defect detection tools called Rx sensors, the program is rolled back to a recent checkpoint. Rx then analyzes the occurring failure based on the failure symptoms and "experiences" accumulated from previous failures, and determines how to apply environmental changes to avoid this failure. Finally, the program reexecutes from the checkpoint in the modified environment. This process may repeat several times, each time with a different environmental change or from a different checkpoint, until either the failure disappears or a timeout occurs. If the failure does not recur in a reexecution attempt, the execution environment is reset to normal to avoid the time and space overhead imposed by some of the environmental changes.

In our idea, the execution environment can include almost everything that is external to the target application but can affect the execution of the target application. At the lowest level, it includes the hardware such as process architectures, devices, etc. At the middle level, it includes the OS kernel such as scheduling, virtual memory management, device drivers, file systems, network protocols, etc. At the highest level, it includes standard libraries, third-party libraries, etc. Such definition of the execution environment is much broader than the one used in previous work [10].

Obviously, the execution environment cannot be arbitrarily modified for reexecution. A useful reexecution environmental change should satisfy two properties. First, it should be *correctness-preserving*, i.e., executing the original program and every step (e.g., instruction, library call and system call) of the program is executed according to the APIs. For example, in the *malloc* library call, we have the flexibility to decide where buffers should be allocated, but we cannot allocate a smaller buffer than requested. Second, a useful environmental change should be able to potentially avoid some bugs. For example, padding every allocated buffer can avoid some buffer overflows from manifesting during reexecution.

Category	Environmental Changes	Potentially-Avoided Bugs	Deterministic?
Memory Management	delayed recycling of freed buffer	double free, dangling pointer	YES
	padding allocated memory blocks	buffer overflow	YES
	allocating memory in an isolated location	memory corruption	YES
	zero-filling newly allocated memory buffers	uninitialized read	YES
Timing-related	scheduling	concurrency bugs	NO
	signal delivery	concurrency bugs	NO
	message reordering	concurrency bugs	NO
User Request Related	dropping user requests	bugs related to the dropped request	Depends

Table 1: Possible environmental changes and their potentially-avoided bugs

Examples of useful execution environmental changes include, but are not limited to, the following categories:

(1)Memory management based: Many software bugs are memory related, such as buffer overflows, dangling pointers, etc. These bugs may not manifest themselves if memory management is performed slightly differently. For example, each buffer allocated during reexecution can have padding added to both ends to prevent some buffer overflows. Delaying the recycling of freed buffers can reduce the probability for a dangling pointer to cause memory corruption. In addition, buffers allocated during reexecution can be placed in isolated locations far away from existing memory buffers to avoid some memory corruption. Furthermore, zero-filling new buffers can avoid some uninitialized read bugs. *Since none of the above changes violate memory allocation or deallocation interface specifications, they are safe to apply.*

(2)Timing based: Most non-deterministic software bugs, such as data races, are related to the timing of asynchronous events. These bugs will likely disappear under different timing conditions. Therefore, Rx can forcefully change the timing of these events to avoid these bugs during reexecution. For example, increasing the length of a scheduling time slice will likely avoid context switches during buggy critical sections.

(3)User request based: Since it is infeasible to test every possible user request before releasing software, many bugs occur due to unexpected user requests. For example, malicious users issue malformed requests to exploit buffer overflow bugs during stack smashing attacks [11]. These bugs can be avoided by dropping some users' requests during reexecution. Of course, since the user may not be malicious, this method should be used as a last resort after all other environmental changes fail.

Table 1 lists some environmental changes and the types of bugs that can be potentially avoided by them. Although there are many such changes, due to space limitations, we only list a few examples for demonstration.

After a reexecution attempt successfully passes the problematic program region for a threshold amount of time, the environmental changes applied during the successful reexecution are disabled to reduce space and time overhead. Furthermore, the failure symptoms and the effects of the environmental changes applied are recorded. This speeds up the process of dealing with future failures with similar symptoms and code locations. Additionally, Rx provides all such diagnostic information to programmers together with core dumps and other basic postmortem bug analysis information.

If the failure still occurs during a reexecution attempt, Rx will rollback and reexecute the program again, either with a different environmental change or from an older checkpoint. For example, if one change (e.g. padding buffers) cannot avoid the bug during the reexecution, Rx will rollback the program again and try another change (e.g. zero-filling new buffers) during the next reexecution. If none of the environmental changes work, Rx will rollback further and repeat the same process. If the failure still remains after a threshold number of iterations of rollback-reexecute, Rx will resort to previous solutions, such as whole program rebooting [14, 25] or micro-rebooting [7, 6, 8], as supported by applications.

Upon a failure, Rx follows several rules to determine the order in which environmental changes should be applied during the recovery process. First, if a similar failure has been successfully avoided by Rx before, the environmental change that worked previously will be tried first. If this does not work, or if no information from previous failures exists, changes with small overheads (e.g. padding buffers) are tried before those with large overheads (e.g. zero-filling new buffers). Changes with negative side effects (e.g. dropping requests) are tried last. Changes that do not conflict, such as padding buffers and changing event timing, can be applied simultaneously.

There is a rare possibility that a bug still occurs during reexecution but is not detected in time by Rx's sensors. In this case, Rx will claim a recovery success while it is not. Addressing this problem requires using more rigorous on-the-fly software defect checkers as sensors. This is currently a hot research area that has attracted much attention. In addition, it is also important to note that, *unlike in failure oblivious computing, this problem is caused by the application's bug instead of Rx's environmental changes*. Environmental changes just make the bug manifest itself in a different way. Furthermore, since Rx logs its every action including what environmental changes are applied and what the results are, programmers can use this information to analyze the bug.

3 Rx Design Overview

While the Rx implementation borrows ideas from previous work, many design issues need to be addressed differently due to differing goals. First, Rx targets software failures instead of hardware failures or OS crashes. Therefore, the checkpointing component does not need to be heavy-weight. Second, Rx does not require deterministic replay. Instead, Rx needs the exact opposite: non-determinism. Therefore, issues such as checkpoint management and the output commit problem [12] need to be addressed differently.

As shown in Figure 2, Rx consists of five components: (1) sensors for detecting failures and bugs, (2) a Checkpoint-and-Rollback (CR) component, (3) a proxy for making server recovery process transparent to clients, (4) environmental wrappers, and (5) a control unit that determines the recovery plan for an occurring failure.

Sensors detect software bugs and failures by dynamically monitoring applications' execution. There are two types of sensors. The first type detects software errors such as assertion failures, access violations, divide-by-zero exceptions, etc. This type of sensor is relatively easy to implement by simply taking over OS-raised exceptions. The second type of sensor detects software bugs such as buffer overflows, accesses to freed memory etc., before they cause the program to crash. This type of sensor leverages existing dynamic bug detection tools, such as our previous work, SafeMem [17], that have low run-time overhead (only 1.6-14%) for detecting memory-related bugs in server programs.

The CR (Checkpoint-and-Rollback) component takes checkpoints of the target application and rolls back the application to a previous checkpoint upon failure. Rx uses a light-weight checkpointing solution that is designed for surviving software failures. At a checkpoint, Rx stores a snapshot of the application into memory. Similar to the fork operation, Rx copies application memory in a copy-on-write fashion to minimize overhead. The details were discussed in our previous work [24]. Performing rollback is straightforward: simply reinstate the program from the shadow process associated with the specified checkpoint. The CR also supports multiple checkpoints and rollback to any of them.

The environment wrapper performs environmental changes during reexecution. We implement different environmental changes in different components. For example, we implement memory management based changes by wrapping the memory allocation library calls. The kernel deals with timing based changes, such as thread scheduling, signal delay, and other asynchronous timing events. The proxy process, which will be described next, manipulates user requests.

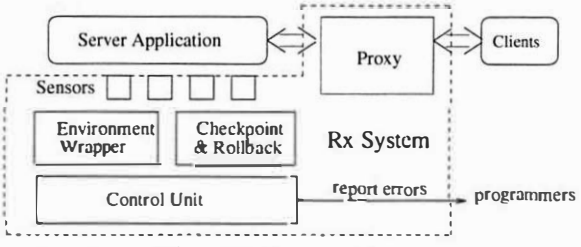


Figure 2: System architecture

To provide the reexecution functionality, Rx uses a proxy to buffer messages between the server and its remote clients. The proxy runs as a separate process to avoid corruption by the server. During normal operation, the proxy simply bridges between the server and its clients, and buffers user requests that are made since the oldest undeleted checkpoint. During a reexecution attempt from a checkpoint, the proxy replays all the user requests received since the checkpoint.

To address the output commit problem, the proxy ensures that every user request is replied to once and only once. For each request, the proxy records whether this request has been answered. If so, a reply made during reexecution is dropped silently. Otherwise, the reply is sent to the corresponding client. In other words, only the first reply goes to the client, no matter whether this first reply is made during the original execution or a successful reexecution attempt.

For applications such as on-line shopping or the SSL hand-shake protocol that require strict session consistency (i.e. later requests in the same session depend on previous replies), Rx can record the signatures (hash values) of all committed replies for each outstanding session, and perform MD5 hash-based consistency checks during reexecution. If a reexecution attempt generates a reply that does not match with the associated committed reply, the session can be aborted abnormally to avoid confusing users.

The control unit analyzes occurring failures and determines which checkpoint to roll back to and which environmental changes to apply during reexecution. After each reexecution, it records the effects (success or failure) into its failure table. This table is used as a reference for future failures and is also provided to programmers for postmortem bug analysis. The control unit also monitors the recovery time and when it exceeds some threshold, it resorts to program restart solutions.

4 Preliminary Results

We have investigated some real, buggy server programs, listed in Table 2. Our analysis shows that these software failures can be dynamically survived by our methods.

In the evaluation, we design four sets of experiments to evaluate different key aspects of Rx: (1) the functionality of Rx in surviving software failures caused by common software defects; (2) the performance overhead of Rx in both server throughput and average response time; (3) how Rx would behave while under malicious attacks

Bugs	Applications	Environment Modification
data race	mysql-4.1.1	change process's priority or make CPU scheduling timeslot longer
buffer overflow	squid-2.3	allocate memory blocks in an isolated address space, or drop request
	apache-2.0.47	
double free	cvs-1.11.4	delay the recycling of recently freed buffers

Table 2: Examples of applications that can benefit from Rx

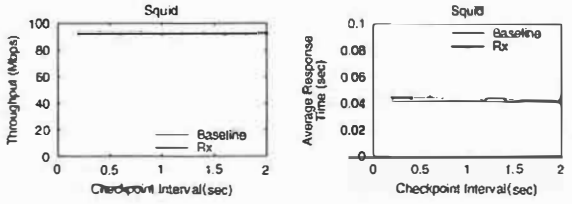


Figure 3: Rx overhead in terms of throughput and average response time for Squid. In these experiments, we do not send the bug-exposing request since we want to compare the pure overhead of Rx with the baseline in normal cases.

that continuously send bug-exposing requests triggering software defects; (4) the benefits of Rx's mechanism of learning from previous failures to speed up recovery.

In particular, Figures 3 shows the overhead of Rx for Squid compared to the baseline (without Rx) for various frequencies of checkpointing. We can see that both throughput and response time are very close to baseline for all tested checkpoint rates. Results for other server applications are similar. In this experiment, we use a workload similar to the one used in [24].

5 Conclusions

In summary, Rx is a non-invasive, informative and safe method for quickly surviving software failures to provide highly available service. It does so by reexecuting the buggy program region in a modified execution environment. It can deal with both deterministic and non-deterministic bugs, and requires little to no modification to applications' source code. Because Rx does not forcefully change programs' execution by returning speculative values, it introduces no uncertainty or misbehavior into programs' execution. Moreover, it also provides additional feedback to programmers for their bug diagnosis. Our preliminary results show that Rx is a viable solution and many server programs should be able to benefit from our approach.

6 Acknowledgments

The authors would like to thank the anonymous reviewers for their invaluable feedback. We appreciate useful discussion with the OPERA group members. This research is supported by IBM Faculty Award, NSF CNS-0347854 (career award), NSF CCR-0305854 grant and NSF CCR-0325603 grant.

REFERENCES

[1] A. Avizienis. The N-version approach to fault-tolerant software. *IEEE TSE*, SE-11(12), 1985.

[2] A. Avizienis and L. Chen. On the implementation of N-version programming for software fault tolerance during execution. In *COMPSAC*, 1977.

[3] J. F. Bartlett. A NonStop kernel. In *SOSP*, 1981.

[4] A. Bobbio and M. Sereno. Fine grained software rejuvenation models. In *IPDS*, 1998.

[5] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM TOCS*, 7(1), Feb 1989.

[6] G. Candea, J. Cutler, A. Fox, R. Doshi, P. Garg, and R. Gowda. Reducing recovery time in a small recursively restartable system. In *DSN*, 2002.

[7] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *HotOS*, 2001.

[8] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. A mirorebootable system – Design, implementation, and evaluation. In *OSDI*, 2004.

[9] CERT/CC. Advisories. http://www.cert.org/advisories/.

[10] S. Chandra and P. M. Chen. Whither generic recovery from application faults? A fault study using open-source software. In *DSN/FTCS*, 2000.

[11] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[12] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A survey of rollback-recovery protocols in message-passing system. Technical report, TR CMU-CS-96-181, Carnegie Mellon Univ., 1996.

[13] S. Garg, A. Puliafito, M. Telek,and K. S. Trivedi. On the analysis of software rejuvenation policies. In *COMPASS*, 1997.

[14] J. Gray. Why do computers stop and what can be done about it? In *SRDS*, 1986.

[15] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *FTCS*, 1995.

[16] E. Marcus and H. Stern. *Blueprints for High Availability*. John Willey & Sons, 2000.

[17] F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *HPCA*, 2005.

[18] B. Randell. System structure for software fault tolerance. *IEEE TSE*, 1(2), Jun 1975.

[19] B. Randell, P. A. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Comput. Surv.*, 10(2), Jun 1978.

[20] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr.. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, 2004.

[21] M. Rinard, C. Cadar, D. Roy, and D. Dumitran. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *ACSAC*, 2004.

[22] D. Scott. Assessing the costs of application downtime. Gartner Group, May 1998.

[23] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. In *USENIX ATC*, 2005.

[24] S. Srinivasan, C. Andrews, S. Kandula, and Y. Zhou. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. In *USENIX ATC*, 2004.

[25] M. Sullivan and R. Chillarege. Software defects and their impact on system availability – A study of field failures in operating systems. In *FTCS*, 1991.

[26] W. Vogels, D. Dumitriu, A. Agrawal, T. Chia, and K. Guo. Scalability of the Microsoft Cluster Service. In *USENIX Windows NT Symposium*, 1998.

[27] W. Vogels, D. Dumitriu, K. Birman, R. Gamache, M. Massa, R. Short, J. Vert, J. Barrera, and J. Gray. The design and architecture of the Microsoft Cluster Service. In *FTCS*, 1998.

[28] Y.-M. Wang, Y. Huang, and W. K. Fuchs. Progressive retry for software error recovery in distributed systems. In *FTCS*, 1993.

When Virtual is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments

Tal Garfinkel Mendel Rosenblum
{talg,mendel}@cs.stanford.edu
Stanford University Department of Computer Science

Abstract

As virtual machines become pervasive users will be able to create, modify and distribute new "machines" with unprecedented ease. This flexibility provides tremendous benefits for users. Unfortunately, it can also undermine many assumptions that today's relatively static security architectures rely on about the number of hosts in a system, their mobility, connectivity, patch cycle, etc.

We examine a variety of security problems virtual computing environments give rise to. We then discuss potential directions for changing security architectures to adapt to these demands.

1 Introduction

Virtual machines allow users to create, copy, save (checkpoint), read and modify, share, migrate and roll back the execution state of machines with all the ease of manipulating a file. This flexibility provides significant value for users and administrators. Consequently, VMs are seeing rapid adoption in many computing environments.

As virtual machine monitors provide the same interface as existing hardware, users can take advantage of these benefits with their current operating systems, applications and management tools. This often leads to an organic process of adoption, where servers and desktops are gradually replaced with their virtual equivalents.

Unfortunately, the ease of this transition is deceptive. As virtual platforms replace real hardware they can give rise to radically different and more dynamic usage models than are found in traditional computing environments.

This can undermine the security architecture of many organizations which often assume predictable and controlled change in number of hosts, host configuration, host location, etc. Further, some of the useful mechanisms that virtual machines provide (e.g. rollback) can have unpredictable and harmful interactions with existing security mechanisms.

Virtual computing platforms cannot be deployed securely simply by dropping them into existing systems. Realizing the full benefits of these platforms demands a significant re-examination of how security is implemented.

In the next section we will elaborate on the capabilities that virtual machines provide, new usage models they give rise to, and how this can adversely impact security in current systems. In section 3 we will explore how virtual environments can evolve to meet these challenges. We review related work in section 4 and offer conclusions in section 5.

2 Security Problems in Virtual Environments

A virtual machine monitor (VMM) (e.g. VMware Workstation, Microsoft Virtual Server, Xen), provides a layer of software between the operating system(s) and hardware of a machine to create the illusion of one or more virtual machines (VMs) on a single physical platform. A virtual machine entirely encapsulates the state of the *guest operating system* running inside it.

Encapsulated machine state can be copied and shared over networks and removable media like a normal file. It can also be instantiated on existing networks and requires configuration and management like a physical machine. VM state can be modified like a physical machine, by executing over time, or like a file, through direct modification.

Scaling Growth in physical machines is ultimately limited by setup time and bounded by an organization's capital equipment budget. In contrast, creating a new VM is as easy as copying a file. Users will frequently have several or even dozens of special purpose VMs lying around e.g. for testing or demonstration purposes, "sandbox" VMs to try out new applications, or for particular applications not provided by their regular OS (e.g. a Windows VM running Microsoft Office). Thus, the total number of VMs in an organization can grow at an explosive rate, proportional to available storage.

The rapid scaling in virtual environments can tax

the security systems of an organization. Rarely are all administrative tasks completely automated. Upgrades, patch management, and configuration involve a combination of automated tools and individual initiative from administrators. Consequently, the fast and unpredictable growth that can occur with VMs can exacerbate management tasks and significantly multiply the impact of catastrophic events, e.g. worm attacks where all machines should be patched, scanned for vulnerabilities, and purged of malicious code.

Transience In a traditional computing environment users have one or two machines that are online most of the time. Occasionally users have a special purpose machine, or bring a mobile platform into the network, but this is not the common case. In contrast, collections of specialized VMs give rise to a phenomenon in which large numbers of machines appear and disappear from the network sporadically.

While conventional networks can rapidly "anneal" into a known good configuration state, with many transient machines getting the network to converge to a "known state" can be nearly impossible.

For example, when worms hit conventional networks they will typically infect all vulnerable machines fairly quickly. Once this happens, administrators can usually identify which machines are infected quite easily, then cleanup infected machines and patch them to prevent re-infection, rapidly bringing the network back into a steady state.

In an unregulated virtual environment, such a steady state is often never reached. Infected machines appear briefly, infect other machines, and disappear before they can be detected, their owner identified, etc. Vulnerable machines appear briefly and either become infected or reappear in a vulnerable state at a later time. Also, new and potentially vulnerable virtual machines are created on an ongoing basis, due to copying, sharing, etc.

As a result, worm infections tend to persist at a low level indefinitely, periodically flaring up again when conditions are right.

That machines must be online in conventional approaches to patch management, virus and vulnerability scanning, and machine configuration also creates a conflict between security and usability. Long dormant VMs can require significant time and effort to patch and maintain. Thus, users either forgo regular maintenance of their VMs, increasing the number of vulnerable machines at a site, or lose the ability to spontaneously create and use machines, eliminating a major virtue of VMs.

Software Lifecycle Traditionally, a machine's lifetime can be envisioned as a straight line, where the current state of the machine is a point that progresses monotonically forward as software executes, configuration changes are made, software is installed, patches are applied, etc. In a virtual environment machine state is more akin to a tree: at any point the execution can fork off into N different branches, where multiple instances of a VM can exist at any point in this tree at a given time.

Branches are caused by undo-able disks and checkpoint features, that allow machines to be rolled back to previous states in their execution (e.g. to fix configuration errors) or re-run from the same point many times, e.g. as a means of distributing dynamic content or circulating a "live" system image.

This execution model conflicts with assumptions made by systems for patch management and maintenance, that rely on monotonic forward progress. For example, rolling back a machine can re-expose patched vulnerabilities, reactivate vulnerable services, re-enable previously disabled accounts or passwords, use previously retired encryption keys, and change firewalls to expose vulnerabilities. It can also reintroduce worms, viruses, and other malicious code that had previously been removed.

A subtler issue can break many existing security protocols. Simply put, the problem is that while VMs may be rolled back, an attackers' memory of what has already been seen cannot.

For example, with a one-time password system like S/KEY, passwords are transmitted in the clear and security is entirely reliant on the attacker not having seen previous sessions. If a machine running S/KEY is rolled back, an attacker can simply replay previously sniffed passwords.

A more subtle problem arises in protocols that rely on the "freshness" of their random number source e.g. for generating session keys or nonces. Consider a virtual machine that has been rolled back to a point after a random number has been chosen, but before it has been used, then resumes execution. In this case, randomness that must be "fresh" for security purposes is reused.

With a stream cipher, two different plaintexts could be encrypted under the same key stream, thus exposing the XOR of the two messages. This could in turn expose both messages if the messages have sufficient redundancy, as is common for English text. Non-cryptographic protocols that rely on freshness are also at risk, e.g. reuse of TCP initial sequence numbers can allow TCP hijacking attacks [2].

Zero Knowledge Proofs of Knowledge (ZKPK), by their very nature, are insecure if the same random nonces are used multiple times. For example, ZKPK authentication protocols, such as Fiat-Shamir authentication [5] or Schnorr authentication [12], will leak the user's private key if the same nonce is used twice. Similarly, signature systems derived from ZKPK protocols, e.g. the Digital Signature Standard (DSS), will leak the secret signing key if two signatures are generated using the same randomness [1].

Finally, cryptographic mechanisms that rely on previous execution history being thrown away are clearly no longer effective, e.g. perfect forward secrecy in SSL. Such mechanisms are not only ineffective in virtual environments, but constitute a significant and unnecessary overhead.

Diversity Many IT organizations tackle security problems by enforcing homogeneity: all machines must run the most current patched software. VMs can facilitate more efficient usage models which derive benefit from running unpatched or older versions of software. This creates a range of problems as one must try and maintain patches or other protection for a wide range of OSes, and deal with the risk posed by having many unpatched machines on the network.

For example, at many sites today users are simply supplied with VMs running their new operating environment and applications are gradually migrated to that environment, or conversely, legacy applications are run in a VM. This can mitigate the need for long and painful upgrade cycles, but leads to a proliferation of OS versions. This makes patch management more difficult, especially in the presence of older, deprecated versions of operating systems.

Virtual machines have also changed the way that software testing takes place. Previously one required a large number of usually dedicated test machines to test out a new piece of software, one for each different OS, OS version (service pack), patch level, etc. Now each developer or tester can simply have their own collection of virtual test machines. Unfortunately, if these machines are not secured they rapidly become a cesspool of infected machines.

Mobility VMs provide mobility similar to a normal file; they can easily be copied over a network or carried on portable storage media. This can give rise to host of security problems.

For a normal platform, the trusted computing base (TCB) consists of the hardware and software stack. In a VM world, the TCB consists of all of the hosts that a VM has run on. Combined with a lack of history, this can make it very difficult to figure out how far a compromise has extended, e.g. if a file server has been compromised, any VM that was on the server may have been backdoored by an attacker. Determining which VMs were exposed, subsequently copied, etc. can be quite challenging.

Similar problems arise with worms and viruses. Infecting a VM is much like infecting a normal executable. Further, direct infection provides access to every part of a machines state irrespective of protection in the guest OS.

Using VMs as a general-purpose solution for mobility [10, 11] poses even more significant issues. Migrating a VM running on someone's home machine of unknown configuration into a site's security perimeter is a risky proposition at best.

From a theft standpoint, VMs are easy to copy to a remote machine, or walk off with on a storage device. Similar issues of proprietary data loss due to laptop theft are consistently cited as one of the largest sources of financial loss due to computer crime [9].

That VMs are such coarse grain units of mobility can also magnify the impact of theft. Facilitating easy movement of one's entire computing environment (e.g. on a USB keychain) makes users more inclined to carry around all of their (potentially sensitive) files instead of simply the ones they need.

Identity In traditional computing environments there is often an ad-hoc identity associated with a machine. This can be as simple as a list of MAC addresses, employee names, and office numbers. Without such mechanisms it can be extremely difficult to establish who is responsible for a machine, e.g. who to contact if the machine turns malicious or who is responsible for its origin/current state.

Unfortunately, these static methods are impractical for VMs. The dynamic creation of VMs makes the use of MAC addresses infeasible. Often VMs just pick a random MAC address (e.g. in VMware Workstation), in the hope of avoiding collisions.

Identifying machines by location/Ethernet port number is also problematic since a VM's mobility makes it difficult to establish who owns a VM running on a particular physical host. Further, there are often multiple VMs on a physical host, thus shutting off the port to a machine can end up disabling non-malicious VMs as well.

Establishing responsibility is further complicated as VMs have more complicated "ownership histories" than normal machines. A specialized virtual machine may be passed around from one user to the next, much

like a popular shell script. This can make it very difficult to establish just who made what changes to get a machine into its present state.

Data Lifetime A fundamental principle for building secure systems is minimizing the amount of time that sensitive data remains in a system [6]. A VMM can undermine this process. For example, the VMM must log execution state to implement rollback. This can undermine attempts by the guest to destroy sensitive data (e.g. cryptographic keys, medical documents) since data is never really "dead," i.e. data can always be made available again within the VM.

Outside the VM, logging can leak sensitive data to persistent storage, as can VM paging, checkpointing, and migration, etc. This breaks guest OS mechanisms to prevent sensitive data from reaching disk, e.g. encrypted swap, pinning sensitive memory, and encrypted file systems.

As a result sensitive files, encryption keys, passwords, etc. can be left on the platform hosting a VM indefinitely. Because of VMs' increased mobility, such data could easily be spread across several hosts.

Similar Problems in Traditional Computing Environments Some existing platforms exhibit security problems similar to those found in virtual environments. Laptops are known for making it difficult to maintain a meaningful network perimeter by transporting worms into internal networks, and sensitive data (e.g. source code) out, thus making the firewall irrelevant. Undo features like Windows Restore introduce many of the same difficulties as rollback in VMs. Transience occurs with dual boot machines, and other occasionally used platforms.

These examples can lend insight into the impact of VMs. However, they differ in a variety of ways. Most of these technologies are deployed in limited parts of IT organizations or see infrequent use; as virtualization is adopted, these dynamic behaviors become the common case. Similar characteristics manifest by other platforms (e.g. mobility, transience) tend to be more extreme in VMs as VMs are software state. Finally, VMs tend to magnify problems with the rapid growth and novel uses they facilitate.

Notably, adapting virtual computing environments to meet these challenges also provides a solution for mobile platforms.

3 Towards Secure Virtual Environments

The dynamic usage models facilitated by virtual platforms demand a dedicated infrastructure for enforcing security policies. We can provide this by introducing a ubiquitous virtualization layer, and moving many of the security and managment functions of guest operating systems into this layer.

Ubiquity allows administrators to flexibly reintroduce the constraints that virtualization relaxes on mobility and data lifetime. Moving security and management functions (e.g. firewalling, virus scanning, backup) from the guest OS to the virtualization layer allows delegation to a central administrator. It also permits management tasks to be automated and performed while VMs are offline, thus aiding issues of usability, scale and transience.

We will briefly outline what such a layer would look like and how it can address the challenges raised in the prior section.

Outlining a Virtualization Layer The heart of a virtualization layer is a high assurance virtual machine monitor. On top of it would run a secure distributed storage system, and components replacing security and management functions traditionally done in the guest OS.

Enforcing policies such as limiting VM mobility and connectivity requires that the virtualization layer on a particular machine be trusted by the infrastructure. Virtualization layer integrity could be verified either through normal authentication and access controls, or through dedicated attestation hardware e.g. TCPA.

Policy at this layer could limit replication of sensitive VMs and control movement of VMs in and out of a managed infrastructure. Document control style policies could prevent certain VMs from being placed onto removable media, limit which physical hosts a VM could reside on, and limit access to VMs containing sensitive data to within a certain time frame.

User and machine identities at this layer could be used to reintroduce a notion of ownership, responsibility and machine history. Tracking information such as the number of machines in an organization and their usage patterns could also help to gauge the impact of potential threats.

Encryption at this layer could help address data lifetime issues due to VM swapping, checkpointing, rollback, etc.

VMM Assurance A VMM's central role is providing secure isolation. The need to preserve this property is sometimes seen as an argument against moving functionality out of the guest operating system. However, such arguments overlook the inherent flexibility available in a VMM. In essence, a virtual machine monitor is nothing more than a microkernel with a

hardware compatibility layer. As such, it can support arbitrary protection models for services running at the virtualization layer.

For example, firewall functionality running outside of a guest OS would be hosted in its own protection domain (e.g. a paravirtualized VM), and could utilize a special purpose operating system affording better assurance, greater efficiency, and a more suitable protection model than common OSes.

Other requirements for building a high assurance VMM (e.g. device driver isolation) have been explored elsewhere [7].

3.1 Benefits

Moving security and management functions out of the guest OS provides a variety of benefits including:

- **Delegating Management**

  A virtual environment provides maximum utility when users can focus on using their VMs however they please, without having to worry about managing them.

  Moving security functionality out of guest OSes makes it easier to delegate management responsibilities to automated services and site administrators. It also obviates the need for homogeneous systems where every machine runs a common management suite (e.g. LANDesk), or where an administrator must have an account on every machine.

  As administrators can externally modify VMs, tasks not moved outside of the VM can still be delegated while VMs are offline. Much of the required scanning, patching, configuration, etc. can be done by a service running on the virtualization layer that would periodically scan and maintain archived VMs.

  In a virtualization layer, VMs are first-class objects, instead of merely a collection of bits (as in today's file systems). Thus, operations that today require reconfiguration could be provided transparently, e.g. users should be able to copy VMs just as they would a normal file, without having to bring them online and reconfigure. The infrastructure could appropriately update hostname, cryptographic keys, etc. to reflect the new machine identity.

  Suspended VMs could be executed in a "sandboxed" environment to allow certain configuration changes to anneal and ensure that they do not break the guest.

- **Guest OS Independence**

  Moving security and managment components to the virtualization layer makes them independent of the structure of the guest operating system. Thus, these components can provide greater assurance, as they can largely specify their own software stack and protection model and are isolated from the guest OS. In contrast, today's host-based firewalls, intrusion detection and anti-virus software are tightly coupled with the fragile monolithic operating systems they try to protect, making them trivial to bypass.

  This flexibility opens the door for the adoption of more secure and flexible operating systems as a foundation for infrastructure services. Further, because the infrastructure can now authenticate and trust components running at network end-points, it can now delegate responsibility to these end-points, thus making policies such as trustworthy network quarantine (i.e. limiting network access based on VM contents) feasible.

- **Lifecycle Independence** Moving security relevant state out of the guest OS solves many difficulties caused by rollback.

  This can be accomplished by moving security mechanisms out of the guest completely, into e.g. an external login mechanism, or by modifying guests to store state such as user account information, virus signatures, firewall rules, etc. in dedicated storage that would operate independent of rollback. A combination of both approaches is likely necessary.

  For protocol related issues, making guest software lifecycle independant is likely the easiest path forward, and seems possible without major changes to today's systems.

  As a first step, lifecycle dependent algorithms could be replaced with lifecycle independent variants, e.g. ZKPK based signature schemes (such as DSS) can be replaced with lifecycle independent signatures schemes such as RSA.

  Guest software must have also some way of being notified when a VM has been restarted, so that it can refresh any keys it is currently holding, perhaps a variation on existing approaches for notifying applications when a laptop has awakened from hibernation. Finally, randomness (e.g. data from Linux's `/dev/random`) should be obtained directly from the VMM instead of relying on state/events within the VM.

- **Securely Supporting Diversity**

  A virtual infrastructure should allow users to use old unpatched VMs with diverse OSes much as they would be able to use old or non-standard files without having to change them. This avoids

problems such as patches breaking VMs and being unable to secure deprecated versions of software where patches are no longer available.

Enforcing policy from outside of VMs facilitates this through the use of vulnerability specific protection as an alternative to software modification. For example, vulnerability specific firewall rules, such as Shields [13], can allow users to run unpatched versions of applications and operating systems while still accessing as much network functionality as is safely possible.

Finally, today greater diversity requires supporting N different versions of security software (e.g. firewall, intrusion detection). While specialized policy is still required for scanning particular OSes, putting management at the virtualization layer eliminates this redundant infrastructure.

There are many challenges to building an architecture that securely allows the full potential of VMs to be realized. However, we believe the direction forward is clear. Moving security relevant functionality out of guest operating system to a ubiquitous virtualization layer provides a more secure and flexible model for managing and using VMs.

4 Related Work

Previous work has examined the security benefits of moving intrusion detection [8], and logging [3, 4] out of the guest e.g. to leverage the isolation and ability to interpose on all system events provided by the VMM. The benefits of trust and flexible assurance provided by placing components such as the firewall outside of the VM [7] have also been explored.

Recent projects have examined how virtualization can enhance manageability [11], mobility [10], and security [3, 4, 7, 8]. Unfortunately, this work has only considered single hosts or assumed an entirely new organizational paradigm (e.g. utility computing), overlooking how virtual machine technology impacts security in current organizations.

Some of the problems presented here are beginning to be addressed by VMware ACE, such as controlling VM copying, preventing the spread of VM contents (encrypted virtual disks and suspend files) and some support for network quarantine.

5 Conclusions

We expect end-to-end virtualization to become a normal part of future computing environments. Unfortunately, simply providing a virtualization layer is not enough. The flexibility that makes virtual machines such a useful technology can also undermine security within organizations and individual hosts.

Current research on virtual machines has focused largely on the implementation of virtualization and its applications. We believe that further attention is due to the security risks that accompany this technology and the development of infrastructure to meet these challenges.

Acknowledgments

This work benefited significantly from discussions with Ben Pfaff and Jim Chow. We are very grateful for generous feedback given to us by Dan Boneh, Jeff Mogul, Armando Fox, Emre Kiciman, Peter Chen, and Martin Casado. This material is based upon work supported in part by the National Science Foundation under Grant No. 0121481.

References

[1] M. Bellare, S. Goldwasser, and D. Micciancio. "Pseudo-random" number generation within cryptographic algorithms: The DDS case. In *CRYPTO*, pages 277–291, 1997.

[2] S. M. Bellovin. Security problems in the TCP/IP protocol suite. *SIGCOMM Comput. Commun. Rev.*, 19(2):32-48, 1989.

[3] P. M. Chen and B. D. Noble. When virtual is better than real. In *(HOTOS-VIII)*, Schloss Elmau, Germany, May 2001.

[4] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *OSDI*, 2002.

[5] U. Fiege, A. Fiat, and A. Shamir. Zero knowledge proofs of identity. In *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 210–217, New York, NY, USA, 1987. ACM Press.

[6] T. Garfinkel, B. Pfaff, J. Chow, and M. Rosenblum. Data lifetime is a systems problem. In *Proc. 11th ACM SIGOPS European Workshop*, september 2004.

[7] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003)*, October 2003.

[8] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[9] L. Gordon, M. L. W. Lucyshyn, and R. Richardson. CSI/FBI computer crime and security survey. `http://www.gocsi.com`, 2004.

[10] M. Kozuch and M. Satyanarayanan. Internet suspend/resume. In *Forth IEEE Workshop on Mobile Computing Systems and Applications*, pages 40–, 2002.

[11] C. Sapuntzakis and M. S. Lam. Virtual appliances in the Collective: A road to hassle-free computing. In *(HOTOS-XI)*, May 2003.

[12] C.-P. Schnorr. Efficient signature generation by smart cards. *J. Cryptology*, 4(3):161–174, 1991.

[13] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proc. of ACM SIGCOMM*, August 2004.

Make Least Privilege a Right (Not a Privilege)

Maxwell Krohn*, Petros Efstathopoulos†, Cliff Frey*, Frans Kaashoek*, Eddie Kohler†,
David Mazières‡, Robert Morris*, Michelle Osborne‡, Steve VanDeBogart† and David Ziegler*
*MIT †UCLA ‡NYU
asbestos@scs.cs.nyu.edu

ABSTRACT

Though system security would benefit if programmers routinely followed the *principle of least privilege* [24], the interfaces exposed by operating systems often stand in the way. We investigate why modern OSes thwart secure programming practices and propose solutions.

1 INTRODUCTION

Though many software developers simultaneously revere and ignore the principles of their craft, they reserve special sanctimony for the *principle of least privilege*, or *POLP* [24]. All programmers agree in theory: an application should have the minimal privilege needed to perform its task. At the very least, five *POLP requirements* must be followed: (1) split applications into smaller protection domains, or "compartments"; (2) assign exactly the right privileges to each compartment; (3) engineer communication channels between the compartments; (4) ensure that, save for intended communication, the compartments remain isolated from one another; and (5) make it easy for anyone to audit the intended separation.

Unfortunately, modern operating systems make these requirements onerous, dangerous, or impossible to apply. In our experience (detailed in Section 2.2), building least-privileged software is cumbersome and labor-intensive: following POLP feels more like an abuse of the operating system's interface than a judicious use of its features. Most programmers spare themselves these difficulties by reverting to monolithic, over-privileged application designs. Such neglect exposes machines to attacks both old and new, from remote attacks on privileged servers to "install attacks" (exploiting users' willingness to run high-privilege installers to infect machines with malware). We cannot write bug-free applications or prevent honest users from occasionally executing malicious code. Instead, our best hope is to contain the damage of evil code by resurrecting POLP.

In this paper, we examine some ways that current OSes discourage development of least-privilege applications (Section 2), then propose OS design ideas that might encourage it instead. A first approximation of a POLP-friendly system is one based on *capabilities*, discussed in Section 3. Though capabilities have historically flummoxed application designers, we present a more familiar interface, based on the Unix file system. In Section 4, we discuss shortcomings in this proposed design: system weaknesses might still allow vulnerabilities to spread, and process-sized compartments are too coarse-grained. We then propose a solution based on *decentralized mandatory access control* [17]. The end result is a new operating system called *Asbestos*.

2 LESSONS FROM CURRENT SYSTEMS

Administrators and programmers can achieve POLP by pushing the features in modern Unix-like operating systems, but only partially, and with important practical drawbacks.

2.1 `chrooting` or `jailing` Greedy Applications

Because Unix grants privilege with coarse granularity, many Unix applications acquire more privileges than they require. These "greedy applications" can be tamed with the `chroot` or `jail` system calls. Both calls confine applications to *jails*, areas of the file system that administrators can configure to exclude setuid executables and sensitive files. FreeBSD's `jail` goes further, restricting a process's use of the network and interprocess communication (IPC). System administrators with enough patience and expertise can `chroot` or `jail` standard servers such as Apache [1], BIND [3] and sendmail [26], though the process resembles stuffing an elephant into a taxicab.

Even when possible, the `chroot/jail` approach faces more fundamental drawbacks:

Jails are heavyweight. The jailed file system must contain copies of system-wide configuration files (such as `resolv.conf`), shared libraries, the run-time linker, helper executable files, and so on. Maintaining collections of duplicated files is an administrative difficulty, especially on systems with many jailed applications.

Jails are coarse-grained. Running a process in a jail is similar to running it on its own virtual machine. Two jailed applications can share files only if one's namespace is a superset of the other, or if inefficient workarounds are used, such as NFS-mounting a local file system.

Jails require privilege. Unprivileged users may not call `chroot` or `jail`.[1] Jails are therefore ill-suited for containing the many untrusted applications that should not have privileges, such as executable email attachments or browser plugins.

Finally, `chroot` or `jail`'s *ex post facto* imposition of security is no substitute for POLP-based design. For example, a typical dynamic content Web server (such as Apache with PHP [18]) runs many logically unrelated scripts within the same address space. A vulnerability in any one script exposes all other scripts to attack, regardless of whether the server is jailed.

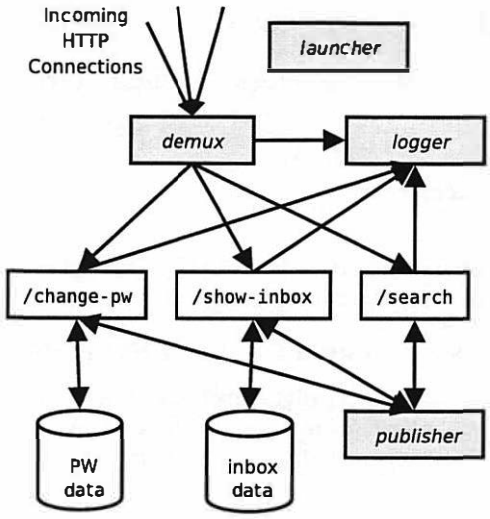


Figure 1: Block diagram of the OKWS system. Standard processes are shaded, while site-specific services and databases are shown in white. The privileged *launcher* process launches the *demux*, *publisher*, *logger* and the site-specific services. The databases shown might either be running locally, or on different machines.

2.2 Ad-Hoc Privilege Separation

True privilege separation is possible on Unix through a collection of ad-hoc techniques. For instance, our POLP-based OK Web Server (OKWS) [12] uses a pool of worker processes to sequester each logical function of the site (e.g. `/show-inbox`, `/change-pw`, and `/search`) into its own address space. The *demux*, a small, unprivileged process, accepts incoming HTTP requests, analyzes their first lines, and forwards them to the appropriate workers using file descriptor passing. Workers then respond to clients directly. A privileged *launcher* process starts this process suite, ensuring that processes are jailed into empty subtrees of the file system, and that they do not have the privileges to interact with one another. Finally, since workers' `chroot` environments prohibit them from accessing the root file system directly, they write HTTP log entries and read static HTML content via small, unprivileged helper processes: the *logger* and the *publisher*, respectively. Figure 1 shows a block diagram of a simple OKWS configuration.

The goal of this design is to separate application logic into disjoint compartments, so that any local vulnerability (especially in site-specific worker processes) cannot spread. In particular, workers cannot send each other signals or trace each other's system calls, they cannot access each other's databases, they cannot alter any executable or library, and they cannot access each other's coredumps. Unfortunately, achieving these natural requirements complicates OKWS. Its launcher must:

1. Establish a `chroot` environment, with the correct file system permissions, that contains the appropriate shared libraries, configuration files, run-time linker, and worker executables.
2. Obtain unused UID and GID ranges on the system.
3. Assign the ith worker its own UID u_i and GID g_i.
4. Allocate a writable coredump directory for each UID.
5. Change the ith worker's executable to have owner `root`, group g_i, and access mode `0410`.
6. Call `chroot`.
7. For each worker process i: kill all processes running as user u_i or group ID g_i; fork; change user ID to u_i and group ID to g_i; `chdir` into the dedicated dump directory; and call `exec` on the correct executable.

The `chown` call in Step 5, the `chroot` call in Step 6, and the `setuid` call in Step 7 all require privileged system access, so the launcher must run as root. Unix offers no guarantees of an atomic UID reservation (as required in Step 2) or race-free file system permission manipulations (as required throughout). Even ignoring these potential security problems, this design requires involved IPC to coordinate worker and helper processes.

Other systems use similar techniques to solve related problems. Examples include remote execution utilities such as OpenSSH [23] and REX [10], and mail transfer agents such as qmail [2] and postfix [21]. Considering these applications and others, a trend emerges: in each instance, the intricate mechanics of privilege separation are invented anew. To audit the exact security procedures of these applications, one must comb tens of thousands of lines of code, each time learning a new system. Even automated tools that separate privileged operations [5] require root access.

2.3 A User-Level POLP Library?

At first glance, a user-level POLP library might seem able to abstract the security-related specifics of applications like OKWS, qmail, and so on. One such example of this approach is found in the Polaris system for Windows XP [30], which applies POLP to virus-prone client applications like Web browsers and spreadsheets[2] via `chroot`-like compartments. Such solutions have three drawbacks. First, they require privileged access to the system. Second, libraries must work around the lack of good OS support for sharing across compartments: since jailed processes work with copies of files, synchronization schemes are required to reconcile copies after changes. (For example, Polaris email plug-ins run in a jail with a copy of the attachment; a persistent "synchronizer" process updates the original if the plug-in changes the copy.) Finally, we suspect that POLP techniques used in more complicated servers such as OKWS do not generalize well. As evidence, both OKWS and REX, an ssh-like login facility, use the same libraries (the SFS toolkit [16]) but share little security-related code. This comes as no surprise since the two have different security aims: OKWS hides most of the file system, while REX exposes it to authorized users; OKWS must support

millions of possible users, while REX serves only those with login access to a given machine; application designers can extend OKWS with site-specific code, while REX runs unmodified. Fitting both application types into one general template seems a tall order.

2.4 Unix as a Capability System

One of the main difficulties with ad-hoc privilege separation is that starting with a privileged process and subtracting privileges is more cumbersome and error-prone than starting with a totally unprivileged process and adding privileges. Unix-like operating systems in general favor the subtractive model, while capability-based operating systems [4, 28] favor the additive one. But Unix file descriptors are in fact capabilities. By hobbling system calls sufficiently-—either through system call interposition [7, 22] or small kernel modifications—we can emulate those semantics of capability-based operating systems that enable privilege separation.

The idea is to allow calls that use already-opened file descriptors (such as `read`, `write`, and `mmap`), but shut off all "sensitive" system calls, including those that create new capabilities (such as `open`), assign capabilities control of named resources (such as `bind`), and perform file system modifications, permissions changes, or IPC without capabilities (such as `chown`, `setuid`, or `ptrace`). In OKWS, the launcher could apply such a policy to the worker processes, which only require access to inherited or passed file descriptors. The launcher could run without privilege, and would no longer navigate the system call sequence seen in Section 2.2. By disabling all unneeded privileges, the operating system could enforce privilege separation by default.

This works because Unix's capability-like system calls are *virtualizable*. Processes are usually indifferent to whether a file descriptor is a regular file, a pipe to another process, or a TCP socket, since the same `read` and `write` calls work in all three cases. In practical terms, virtualization simplifies POLP-based application design. Splitting a system into multiple processes often involves substituting user-space helper applications for kernel services; for instance, OKWS services write log entries to the *logger* instead of a Unix file. With virtualizable system calls, user processes can mimic the kernel's interface; programmers need not rewrite applications when they choose to reassign the kernel's role to a process.

More important, virtualizable system calls enable *interposition*. If an untrustworthy process asks for a sensitive capability, a skeptical operator can babysit it by handing it a pipe to an interposer instead. The interposer allows harmless queries and rejects those that involve sensitive information. If the kernel API is virtualizable, then the operator need not even recompile the untrustworthy process to interpose on it.

Unfortunately, most Unix system calls resist virtualization. Some do not involve any capability-like objects; others use hard-wired capabilities hidden in the kernel, such as "current working directory" and "file system root". User-level emulation of these problematic calls—which include `open`—is messy, if not impossible; but scrapping `open` in the name of POLP seems unlikely to compel the average programmer.

3 Operating System Support for POLP

With the lessons from Unix, we can imagine a POLP-friendly operating system interface, in which all system calls are capability-based and virtualizable like `read` and `write`. Adding universal virtualization support to a Unix-like capability system would cover all five POLP requirements. With capabilities, application programmers can split their program into isolated compartments (#1 and #4), granting each compartment only the privileges necessary to complete its task (#2). With virtualization, programmers use standard interfaces and libraries for communication between these compartments (#3), and auditors can understand this communication by interposing at the interfaces (#5). This section presents a hypothetical design for such a system, which we'll call *Unestos*.

3.1 Unestos Design

In Unestos, interactions between a process and other parts of the system take the form of *messages* sent to *devices*. Devices include processes and system services as well as hardware drivers. Messages follow the outline "perform operation O on capability C, and send any reply to capability R." The kernel forwards this message to the device that originally issued C. There are a small number of operation types, as in NFS [25] and Plan 9's 9P [19]: `LOOKUP`, `READ`, `WRITE`, and so forth. The message types and their associated syntax are conventions; the kernel only enforces or interprets those messages sent to kernel devices. Requests and replies are sent and received asynchronously.

This design aids virtualization. All of a process's interactions with the system—whether with the kernel or other user applications—take the same form, explicitly involve capabilities, and shun implicit state. Consider, for example, the Unix call `open("foo")`. This call in Unestos would translate to a message that a process P sends to the file server device *FS*:

$$P \rightarrow \langle C_{\text{CWD}}, \text{LOOKUP}, \texttt{"foo"}, C_P \rangle \rightarrow FS.$$

The first argument is a capability C_{CWD} that identifies P's current working directory. The second is the command to perform, the third represents the arguments, and the fourth is the capability to which the file system should send its response. Since Unestos makes explicit the CWD state hidden in the Unix system call, either the file server or a user process masquerading as the file server can answer the message.

3.2 Naming and Managing Capabilities

When an Unestos process P_1 launches a child process P_2, it typically grants P_2 a number of capabilities, rang-

ing from directories on the file system to opened network connections. How can P_2 then access these capabilities? Traditional capability systems such as EROS favor global, persistent naming, but persistence has proven cumbersome to kernel and application designers [27].

Instead, we advocate a per-process, Unix-style namespace. Under Unestos, P_1 makes capabilities available to P_2 as files in P_2's namespace. Suppose P_1's namespace contains a tree of files and directories under `/secret`, and P_1 wishes to grant P_2 access to files under `/secret/bob`. As in Plan 9 [20], P_1 can mount `/secret/bob` as the directory `/home` in P_2's namespace. Unlike in Plan 9, the state implicit in the per-process namespace is handled at user level, and the kernel only traffics in messages sent to capabilities. For example, when the process P_2 opens a file under `/home`, the user level libraries translate the directory `/home` to some capability C. The kernel sees a LOOKUP message on C.

3.3 OKWS Under Unestos

We now consider what OKWS might look like on Unestos. Similar to before, the application suite consists of a *launcher*, *demux* and worker processes. Under Unestos, the logger process simply enforces append-only access to a log file, and might be useful for many applications (much like `syslogd` on today's systems). No publisher process is needed.

The launcher starts each process with an empty namespace (and thus no capabilities), then augments their namespaces as follows:

- In the *logger*'s namespace, mounts a logfile on `/okws/log`.
- In the *demux*'s namespace, mounts TCP port 80 on `/okws/listen`. For each worker process i, makes a socket pair and connects one end to `/okws/worker/`i.
- In worker process i's namespace, mounts the other end of the above socket pair to `/okws/listen`. Mounts a connection to the logger on `/okws/log`. Mounts a read-only capability to the root HTML directory on `/www`.
- In all namespaces, makes required shared libraries available under `/lib`.

The launcher then launches all processes as before.

Under Unix, the launcher had to carefully construct jails, physically copying over files and invoking custom helper applications like the publisher and logger to limit file system access. Unestos, by contrast, lets the launcher expose capabilities to child processes at arbitrary points in their namespaces. Each child receives a synthetic file system perfectly suited to its task.

Moreover, all capabilities available to the Unestos OKWS processes are virtualizable. Workers accept connections on `/okws/listen` regardless of whether they originate from the kernel's TCP stack or the *demux*. Similarly, logging might be to a raw file or through a logging process that enforces append-only behavior; worker processes are oblivious to the difference.

3.4 Discussion

So far, the proposed system features no individually novel ideas; rather, it finds a new point in the OS design space amenable to secure application construction. Similar effects might be possible with message-passing microkernels, or unwieldy system call interposition modules. But in Unestos, the security primitives are few and simple, for both the kernel and application developer. Although the interface exposed to applications feels like the familiar Unix namespace (with added flexibility for unprivileged, fine-grained jails), an application's system interactions are entirely defined by its capabilities, and Unestos behaves like a capability system for the purposes of security analysis.

4 Fine-Grained POLP with MAC

Though we believe Unestos is an improvement over the status quo, it still falls short of enabling the high-level, end-to-end security policies we seek. Applications in Unestos can only express security policies in terms of *processes*, but processes often access many different types of data on behalf of different users. A security policy based on processes alone can therefore conflate data flows that ought to be handled separately. For example, OKWS on Unestos achieves the policy that data from a `/change-pw` process cannot flow to a corrupted `/show-inbox` process; but the policy says nothing about whether user U's data within `/show-inbox` can flow to user V, meaning an attacker who compromises `/show-inbox` might be able to read an arbitrary user's private e-mail.

Of course, a much better policy for OKWS would be that "only user U can access user U's private data". We would like to separate users from one another, much as we separated services in Section 3. Though a user session involves many different processes (such as the *demux*, databases[3], and worker processes), a policy for separating users should be achievable with a few stanzas of privileged code. This section extends Unestos to a new system, *Asbestos*, whose kernel uses flexible mandatory access control primitives to enforce richer end-to-end security policies. We are currently designing and building Asbestos as a full operating system for x86 machines.

4.1 Complete Isolation

One possible approach to better isolation, which we call *complete isolation*, is to prohibit server-side processes from speaking for multiple users. The server must be prepared to run a process for every service–user pair; trusted code in *demux* would route traffic accordingly, and various isolation schemes (such as capabilities) could prevent these processes from communicating. More drastic

separation is possible with virtual machines [11, 32] so that each machine can only speak for one user.

Complete isolation has several drawbacks. First, scalability is a challenge: a process for each service–user pair implies either a CPU-intensive fork-accept-exit model or a memory-intensive large server pool. Second, with no kernel support for tracking data flow, processes are completely responsible for their own access-control checks. The initial check happens at *demux*; a subsequent check is required when each per-user process accesses the database. With each additional process that speaks on behalf of multiple users comes additional access control checks. If application programmers forget or misapply any of these checks, the system can leak sensitive data to attackers.

Finally, complete isolation fails if two processes that were intended to be isolated from each other can communicate with any common third process. The system therefore implicitly trusts all running processes to refrain from enabling unintended communication.

4.2 Decentralized, Fine-Grained MAC

A more principled, and reliable approach to managing data flow is possible with mandatory access control (MAC). The Asbestos operating system proposes a decentralized, fine-grained version of MAC to solve the security problems inherent in an OKWS-like system. Similar to traditional MAC, Asbestos assigns devices on the system to *compartments*, which form a partially-orderable lattice. If device A sends device B a message, and they are in the same compartment, they remain so after delivery. If A's compartment is strictly higher than B's, then receiving a message from A pushes B into A's compartment. If A and B's compartments are incomparable, or A's compartment is strictly less than B's, then message delivery fails. With compartments, Asbestos tracks all devices that have accessed a given datum, whether directly or via proxy.

We propose two important modifications to traditional MAC-based operating systems. First, decentralization [17]: processes can create their own compartments on the fly, so that a Web server can associate each remote user with her own compartment. Second, compartments apply at the fine-grained level of individual memory pages, so that a single process can act on behalf of mutually distrustful users without fear of leaking data among them. Taken together, these two modifications allow application designers to dynamically partition a process's virtual address space into compartmentalized *sub-processes*.

Under Asbestos, OKWS behaves as follows: *demux* peaks into user U's incoming TCP connection, authorizing U based on session state or login information in the HTTP headers. If U is logging on for the first time, *demux* creates a compartment for U; if U is returning, then *demux* reassigns U to its previous compartment. It then forwards U's connection to the appropriate sub-process of the appropriate worker. When handling U's request, the sub-process can access virtual memory pages and devices available to U's compartment; for instance, it might access session state cached on the worker process or a database process trusted to store data for all users. If the sub-process errantly accesses data in V's compartment, the read or write will fail, since U and V occupy incomparable compartments.

Once a worker has finished serving U, it can restore its memory and register state to a saved checkpoint, and is then safe to enter a different sub-process, and speak on behalf of a different user. Finally, since *demux* created the user compartments, it can sanction trusted *declassifiers* to traverse them. For example, it might authorize a trusted statistics collector to comb all pages in a worker's virtual address space, regardless of compartment.

5 Related Work

Asbestos proposes the marriage of previous ideas in systems: the capability-based operating system [4, 13, 28, 33], the per-process name space [20], the virtualizable kernel interface (the logical extension of system-call interposition libraries [7, 22]), and decentralized MAC [17].

Naturally, other operating systems predating Asbestos meet related design goals or offer similar features. Message-based operating systems such as L4, Amoeba, V, Chorus and Spring can isolate system services by running them as independent, user-level processes and provide natural support for interposition through message-based interfaces [14]; Trusted Mach in particular views message-passing from a security perspective [6]. But ports in microkernel systems are coarse as capabilities go; for instance, a process can have a capability for the file server but not for a particular directory. For POLP, application programmers need arbitrary collections of specific capabilities; in this respect, the microkernels of yesteryear do not fit the bill.

The Flask System applies MAC to the Fluke Microkernel [29]. Many of Flask's design principles have found a modern incarnation in SELinux [15], which, like TrustedBSD [31], adds mandatory access control to popular Unix systems. In both, static policy files dictate which resources applications might access, and how processes can interact with one another. Such systems are attractive because they preserve the POSIX interface to which many programmers are accustomed. However, their policy extension model, which is based on privileged files and kernel modules, appears to fall short of the uniformly-analyzable policy extensions decentralized labels can support.

Type safety is another way to enforce operating system security. Coyotos combines capabilities with language-level verification techniques [27]. Singularity combines strong isolation with a type-safe ABI [8]. At user level, the Java Sandbox uses customizable policies to specify an applet's access rights; dynamic sandboxing

shows these policies can be automatically produced [9].

6 Conclusion

Asbestos aims to combine decentralized MAC and capabilities to make POLP convenient, practical and effective for applications like OKWS. We have no proof that other applications would similarly benefit from Asbestos, but we are optimistic. Asbestos provides simple, flexible, and fine-grained mechanisms for achieving the five important POLP requirements without sacrificing performance.

Acknowledgments

The authors thank Lee Badger, Butler Lampson, Mike Walfish and the reviewers. This work was supported by DARPA grants MDA972-03-P-0015 and FA8750-04-1-0090, and by joint NSF Cybertrust/DARPA grant CNS-0430425. David Mazières and Robert Morris are supported by Sloan fellowships.

References

[1] The Apache Software Foundation. Apache. `http://www.apache.org`.

[2] D. J. Bernstein. qmail. `http://cr.yp.to/qmail.html`.

[3] Internet Systems Consortium. Berkeley Internet Name Daemon. `http://www.isc.org/sw/bind`.

[4] A. C. Bomberger, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The KeyKOS nanokernel architecture. In *USENIX Workshop on Microkernels and Other Kernel Architectures*. USENIX, 1992.

[5] D. Brumley and D. X. Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, pages 57–72. USENIX, 2004.

[6] T. Fine and S. E. Minear. Assuring distributed trusted mach. In *Proceedings of the 1993 IEEE Symposium on Security and Privacy*, page 206, Washington, DC, USA, 1993. IEEE Computer Society.

[7] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium*, San Jose, CA, USA, 1996.

[8] G. C. Hunt and J. R. Larus. Singulairty design motivation. Technical Report MSR-TR-2004-105, Microsoft Corporation, Dec. 2004.

[9] H. Inoue and S. Forrest. Anomaly intrusion detection in dynamic execution environments. In *NSPW '02: Proceedings of the 2002 workshop on New security paradigms*, pages 52–60. ACM Press, 2002.

[10] M. Kaminsky, E. Peterson, D. B. Giffin, K. Fu, D. Mazières, and M. F. Kaashoek. REX: Secure, extensible remote execution. In *Proceedings of the 2004 USENIX*, pages 199–212, Boston, MA, June–July 2004. USENIX.

[11] P. Karger, M. Zurko, D. Bonin, A. Mason, and C. Kahn. A retrospective on the VAX VMM security kernel. *Transactions on Software Engineering*, 17(11):1147–1165, 1991.

[12] M. Krohn. Building secure high-performance web services with OKWS. In *Proceedings of the 2004 USENIX*, Boston, MA, June–July 2004. USENIX.

[13] H. Levy. *Capability-based Computer Systems*. Digital Press, 1984.

[14] J. Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, 1996.

[15] P. Loscocco and S. Smalley. Meeting critical security objectives with security-enhanced linux. In *Proceedings of Ottawa Linux Symposium 2001*, June 2001.

[16] D. Mazières. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX*, pages 261–274. USENIX, June 2001.

[17] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 129–142, Saint-Malo, France, October 1997. ACM.

[18] PHP: Hypertext processor. `http://www.php.net`.

[19] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[20] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The use of name spaces in Plan 9. In *Proceedings of the 5th ACM SIGOPS Workshop*, Mont Saint-Michel, 1992.

[21] Postfix. `http://www.postfix.org`.

[22] N. Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, pages 257–271, Washington, DC, August 2003.

[23] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium*, Washington, D.C., August 2003.

[24] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sept. 1975.

[25] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer 1985 USENIX*, pages 119–130, Portland, OR, 1985. USENIX.

[26] The Sendmail Consortium. Sendmail. `http://www.sendmail.org`.

[27] J. S. Shapiro, M. S. Doerrie, E. Northup, S. Sridhar, and M. Miller. Towards a verified, general-purpose operating system kernel. In G. Klein, editor, *Proc. NICTA Formal Methods Workshop on Operating Systems Verification*, Sydney, Australia, 2004. NICTA Technical Report 0401005T-1, National ICT Australia.

[28] J. S. Shapiro, J. Smith, and D. J. Farber. EROS: a fast capability system. In *Proc. Symposium on Operating Systems Principles*, pages 170–185, 1999.

[29] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The flasksecurity architecture: System support for diverse security policies. In *In Proceedings of the Eighth USENIX Security Symposium*, August 1999.

[30] M. Stiegler, A. H. Karp, K.-P. Yee, and M. Miller. Polaris: Virus safe computing for windows XP. Technical Report HPL-2004-221, December 2004.

[31] R. N. M. Watson. TrustedBSD: Adding trusted operating system features to FreeBSD. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 15–28. USENIX Association, 2001.

[32] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[33] M. V. Wilkes and R. M. Needham. *The Cambidge CAP Computer and its Operating System*. North Holland, 1979.

Notes

[1] Were it not for this prohibition, unprivileged users could use control of the `chrooted` top-level directory to elevate privileges. The attack is to make a new directory `/tmp/foo`, hard link from `/tmp/foo/su` to the system `su`, write a new password file `/tmp/foo/etc/passwd`, call `chroot` on `/tmp/foo`, and then call `su` from within the jail.

[2] Polaris appears not as well-suited for larger servers.

[3] We assume for simplicity that databases run locally, though all concepts discussed can generalize to distributed deployments.

Access Control in a World of Software Diversity

Martin Abadi[1], Andrew Birrell[2], and Ted Wobber[2]

[1]*University of California, Santa Cruz*
[2]*Microsoft Research, Silicon Valley*

Abstract

We describe a new design for authentication and access control. In this design, principals embody a flexible notion of authentication. They are compound principals that reflect the identities of the programs that have executed, even those of login programs. These identities are based on a naming tree. Our access control lists are patterns that recognize principals. We show how this design supports a variety of access control scenarios.

1. Introduction

A central concern in securing a computer system is access control: deciding whether to permit a particular form of access to some of the system's resources or data. Classically, the system controls access by using a "reference monitor": a trusted piece of code that is used to make all access decisions [1]. The reference monitor is presented with the identity of a principal that makes a request, the identity of an object (system resource or data protected by the system), and the specific form of access desired. The reference monitor then makes the access control decision by deciding whether to accept the proffered identity, and by consulting access control information associated with the object. The access control function is a predicate that maps principal, object, and operation to a Boolean outcome.

In this paper we consider how to design this access control machinery for a single-host, non-distributed operating system such as Windows or Linux. A direction for our future work will be to extend this design to include distributed systems. For this paper, we ignore issues of compatibility with previous access control machinery.

In the classic design for this purpose, each principal is identified by a small identifier (an SSID in Windows, a user ID in Unix-based systems). The access control data for an operation is an access control list kept with each object, and takes the form of a set whose members are either principals or identifiers for groups. A group, in turn, is a set whose members are either principals or identifiers for further groups. Access is permitted or denied on the basis of the presence of the proffered principal in the closure of the access control list and its constituent groups. (In Windows the group memberships of a principal are actually determined at login time and cached in a token. The semantics are as described above, but the timing is somewhat different: some of the reference monitor's work was done at login time.)

The classic design, unfortunately, has many limitations and drawbacks. These have become increasingly critical in recent years as the diversity of the programs installed in our systems, and of the attacks upon them, have increased. The three drawbacks that we attempt to address in the current design are as follows.

First, the notion that the principal is identified solely with a logged-in user doesn't allow us to express important real-world security situations. The actual user of course isn't really the entity making the access request. The request is being made by a program. The classic design assumes that every program executing in a user's session is acting on the user's behalf and with the user's full trust. That might have been true historically, but it is certainly not true today. For example, the user most likely is happy if Microsoft Word is performing operations on objects that are Microsoft Word documents, but would be unhappy if some ad-ware program was doing so. Similarly, the user might reasonably object if Microsoft Word was spontaneously accessing the user's Quicken database. So we desire that the principal presented to the reference monitor includes some notion of the program that is executing, and also of the program that provoked that execution, and so on back through the execution history.

Second, the classical notion of "logged-in" is inflexible. It is all or nothing, and implies that all mechanisms for authenticating a user are equally trusted. Equivalently, it requires that all authentication mechanisms are part of the trusted computing base. To support a modern execution environment, where

principals might arise from a console login, from a remote terminal login, or from the creation of a background service, batch job, or daemon, and where authentication might be by password, X.509 certificate, smart card, or by an *ad hoc* decision by an application, we require that these circumstances can be included as part of the identity of the principal presented to the reference monitor. This is a prerequisite to permitting the monitor to base its decisions partly on *how* a principal was authenticated.

Finally, once we have included so much extra information within the idea of principal, it becomes untenable to say that the access control data is just a set of principal identifiers. We must be able to express in the access control data a wide variety of constraints on the acceptable principals, based on the wider variety of information now included in our principals' identities. However, in order to maintain any real security, the policies that can be described by this more general mechanism must be expressed in a sufficiently simple language that they can be understood by the people responsible for them.

2. Previous Work

Within the confines of a short paper, we cannot come close to doing justice to the wide range of proposals that have been made to address some of the problems identified in the introduction.

Many writers (and some writers many times) have proposed authentication schemes that go well beyond the basic notion of logged-in user [2,4,7,9]. Most commonly, such schemes allow a principal to adopt a "role" or "restricted context" with the intention of reducing or enhancing the principal's privileges. Some schemes become quite elaborate, including in the resulting compound principals such details as the principal that signed the certificate proving the identity of an executing program. Such designs provide great power, but with a lot of complexity.

Current Java security mechanisms [5] take some account of program execution history by using stack inspection [8] when making access decisions.

Several systems, including current versions of Windows and most current Unix-based systems, support extensibility in their authentication mechanisms. There are a few specialized systems that have made their access control decisions dependent on how the principal was authenticated [3], but this hasn't made its way into the access control machinery of general-purpose operating systems. We believe that this information can be included and used without adding undue complexity to the design.

Other designs that involve compound principals have also resulted in revisions to the design of access control lists, though in somewhat different ways than the present design. For example, the Taos work included access control lists that expressed some logic about which principals match [6].

3. Our Design

Our design is intended to address the deficiencies described above. Specifically, we want to consider in our access control decisions the identity of the authenticated user, the identity of the agency that performed the authentication, and the identity of the program invocations that have brought the computation to its current point. Then we want to have access control lists that allow us to express succinctly and intelligibly a wide variety of commonly useful access control policies.

As will be seen below, the key aspects of our design are:

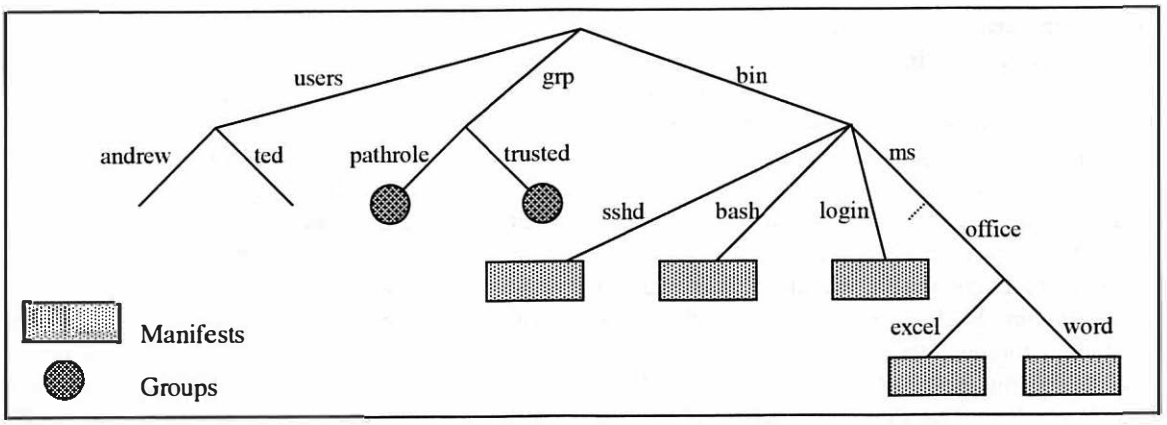
- separation of principal names from the policies and mechanisms that led us to trust those names (the "naming tree");
- compound principals formed by two operators that represent authentication and program invocation; and
- an expressive but straightforward access control list mechanism.

3.1. The Naming Tree

The naming tree is a singly rooted tree in which each arc is labeled with a simple string. Some of the nodes in the tree have attached to them a data structure called a "manifest". A manifest specifies a particular executable "program", by providing the file names and cryptographically secure fingerprints of the constituent parts of the program — its executable, shared libraries, data resources, and so forth. Since we want the identity of an invoked program to be part of a principal name, program invocation is a security-related operation, and we require that programs are named by paths through the naming tree.

The naming tree is also used to name users, and to name groups whose contents can be referenced during the evaluation of access control lists.

Our use of this naming tree lets us separate the mechanisms and policy for constructing the tree from the mechanisms and policy for running a reference monitor. Both are important parts of the overall security



An example naming tree

machinery, but the separation greatly simplifies the authentication and access control mechanisms.

We expect quite familiar mechanisms would be used to construct the tree, though we give no details here. For example, the decision to install a program purporting to be Microsoft Word would likely require a trusted party (such as an authenticated administrator) to inspect certificates (such as X.509 certificates) and agree that the proffered bits really deserve to be given such a trusted name. Once that decision has been made, the presence of the resulting manifest at the node named, e.g., "/bin/ms/office/word" makes the administrator's decision clear, and we can use this in subsequent authentication and access control decisions.

Most likely, the naming tree would have its own access control lists attached to it, to specify which principals can modify which parts of the tree. One advantage of using a tree structure to represent names is that simple policies, for example, that a software publisher controls the namespace beneath it, can be applied. Similarly, the tree structure helps avoid naming conflicts in an ever-evolving namespace.

3.2. Principal Names

A principal name is a string constructed from arcs in the naming tree and operators "/", "@", and "+" according to the following grammar.

- Manifest Name: MN = "/" Arc | MN "/" Arc
- Role: R = "/" Arc | R "/" Arc
- Manifest Role: MR = MN | MR "@" R
- Principal: P = MR | P "+" MR

The system provides exactly two operations that affect principals:

- InvokeProcess(MN)
- ForkRole(R)

"InvokeProcess" runs a program. Its argument "MN" is a manifest name, which is a path from the root of the naming tree to the manifest of the desired program. The system finds the named manifest, loads the appropriate data into a new security context (process, say), and initiates its execution. When the principal that calls InvokeProcess is "P", then the new security context runs as principal "P+MN".

In other words, occurrences of the "+" operator within a principal name represent the history of program invocations that resulted in the currently executing program.

There is one variation of InvokeProcess. A manifest might have been marked as a "service", in which case the new security context runs as the principal "MN", independently from its invoker.

"ForkRole" runs the same program as calls it, but in a new security context with new program state. Its argument "R" is an absolute path in the naming tree. (Role names that are relative to a manifest name are also possible; we do not discuss them here in the interest of brevity.) When the principal that calls ForkRole is "Q", then the new security context runs as principal "Q@R".

In other words, occurrences of the "@" operator within a principal name indicate where a program has decided to adopt a distinguished role. This indication says nothing about whether the role is more or less privileged — that has meaning only to the extent that access control lists grant more or less access to the new principal name.

One critical use of ForkRole is to indicate when a program makes an authentication decision. For example, the system might run a console login program

by invoking the manifest "/bin/login" as a service, thus executing as principal "/bin/login". When the console login program has received a satisfactory user name "andrew" and password, it will use ForkRole to start running itself as "/bin/login @ /users/andrew", then use InvokeProcess to run Andrew's initial command shell "/bin/bash", which will then be executing as the principal "/bin/login @ /users/andrew + /bin/bash".

Similarly, we might run the manifest "/bin/sshd" to listen for incoming SSH connections. After satisfactory authentication through the normal SSH public-key mechanisms it might adopt the role "/bin/sshd @ /users/andrew" then run the command shell, which would execute as "/bin/sshd @ /users/andrew + /bin/bash".

In these two scenarios, if Bash decides to run "cat" (whose manifest is named "/bin/cat") and cat tries to open a file, we would have an access request to the file system from either the principal "/bin/login @ /users/andrew + /bin/bash + /bin/cat" or the principal "/bin/sshd @ /users/andrew + /bin/bash + /bin/cat" respectively. The reference monitor for the file system would then consult the access control list on the requested file to decide whether the given principal should be granted access.

Another example of the utility of roles arises in the context of program installation. Suppose that there is an installer program "/bin/install" that manages the installation of new software. It would be natural for such a program, having checked that it is installing certified Microsoft software, to adopt the role "/bin/installer @ /bin/ms". Acting in this role, the installer might gain permission to update the naming tree under "/bin/ms" (as well as other related system resources), but without having rights to resources designated for other publishers.

Nowhere in these scenarios has the system trusted any of the programs involved: login, sshd, bash, cat, or install. All the system did was to certify the program invocations involved, and that, for example, /bin/login and /bin/sshd chose to adopt the role "/users/andrew". In this design trust occurs only in constructing the naming tree (trusting that the programs really deserve their given names) and as a result of the way in which we write access control lists (which embody our access control decisions).

3.3. Access Control Lists

With complex principal names such as those we propose above, having an access control list ("ACL") be merely a list (or set) of principal names does not give us nearly enough convenience and expressive power. For example, we might want to give access to a user while executing some of a particular set of programs, or when authenticated by some particular set of programs (e.g., /bin/login or /bin/sshd, but not /bin/ftpd); or we might want to give access to a program regardless of its user. While we could perhaps list all allowed principals, that would be awkward at best. Instead we use patterns that recognize principal names.

The exact pattern recognition language that we use is not critical to this idea, although the choice of language will certainly have an impact on the usability of the design, and therefore on the security of the resulting systems. We present here a recognizer for a specialized subset of regular expressions. Obviously, more or less complex recognizers are possible, allowing the expression of more or less complex access control policies.

An ACL is a string constructed from arcs in the naming tree and operators, as follows:

- Atom = Arc | "/" | "@" | "+"
- Item = Atom | "." | "(" ACL ")" | Item "*" | "{" GroupName "}"
- GroupName = "/" Arc | GroupName "/" Arc
- Seq = Item | Seq Item
- ACL = Seq | ACL "|" Seq

The matching rules are similar to those for conventional regular expressions:

- any Atom matches itself;
- "." matches any single Arc (explicitly excluding "/", "@", and "+");
- "(ACL)" matches ACL;
- "Item *" matches zero or more sequential occurrences of Item (greedily);
- "{ GroupName }" matches whatever is matched by the ACL that is the contents of the node GroupName in the naming tree;
- "Seq Item" matches Seq followed immediately by Item;
- "ACL | Seq" matches either ACL or Seq.

A principal "P" matches an ACL "A" iff the string P matches the regular expression that is the contents of A. The match must be complete — all of P, not just a sub-string of it.

"GroupName" provides a mechanism for sharing parts of the recognition machinery amongst multiple ACLs. We place groups within the same naming tree as manifests and role names, with the same assumption that their presence there reflects a trust decision made by a suitable administrator. Recursively defined groups are not permitted.

A reference monitor will grant P its requested access to an object iff P matches the relevant ACL. In doing so, the reference monitor is just performing string manipulation on the principal name and the ACL contents — it doesn't need to use the naming tree itself, except to read referenced groups. (We do not consider here details of controlling access modes, such as "read" and "write"; the reference monitor will of course grant P only the appropriate mode.)

4. Usage Examples

Assume that the naming tree contains the following two groups:

- /grp/pathrole = (/.)* (@ (/.)*)*
- /grp/trusted = (/bin /login | /bin/sshd)

The group "/grp/pathrole" matches a pathname with an arbitrary sequence of roles. The group "/grp/trusted" matches either of a pair of trusted authentication programs.

The following ACL is similar to the baseline semantics of existing systems, that is, it gives access to an explicitly named user, if authenticated by a trusted program:

{/grp/trusted} @ /users/ted (+ {/grp/pathrole}) *

More precisely, the above ACL permits access from any program invoked (directly or indirectly) from one of our trusted authentication programs, provided that the authentication program has adopted the role "/users/ted". In contrast with existing systems, however, the choice of which authentication programs should be trusted is made in the ACL. We could trust different sets of authentication programs for different objects, for different users, or for different access modes.

Our next example is similarly simple, but not at all like traditional access control: it gives access from any of a specific set of programs — those found in the naming tree under /bin/ms/office — regardless of the user who invoked them:

({/grp/pathrole} +) * /bin/ms/office {/grp/pathrole}

One might use such an ACL, for example, to allow Microsoft Office applications to access some auxiliary files, regardless of who is running the applications, while preventing users from doing anything else with the auxiliary files.

Our final example gives access for user "ted" when authenticated by sshd, but only when running some chain of programs with the last one being Microsoft Word:

/bin/sshd @ /users/ted (+ {/grp/pathrole})* +
/bin/ms/office/word

5. Conclusion

Our design has several important aspects that work well together. First, the naming tree lets us separate the policy and mechanisms for certifying programs and groups from the day-to-day authentication and access control mechanisms. Second, we provide just two operators for composing principals, providing expressiveness while retaining simplicity. Third, we use these principals to avoid requiring that the system trust particular authentication programs. Finally, we generalize ACLs to be pattern recognizers, thereby allowing compact expression of sophisticated access control decisions that make full use of the expressiveness of our principals.

We believe that this design allows for authentication and access control in a modern operating system, suitable for the more stringent requirements of a modern security posture in a world with diverse software.

6. Acknowledgements

This work was done at Microsoft Research in the context of the Singularity research project led by Galen Hunt and Jim Larus. In particular, Úlfar Erlingsson and Dan Simon made significant contributions to our many discussions about this design.

References

1. Anderson. "Computer Security Technology Planning Study Volume II", ESD-TR-73-51, Air Force Systems Command,Oct. 1972.
2. Badger et al. "A Domain and Type Enforcement UNIX Prototype". USENIX Comp. Sys., 9(1): 47-83, Winter 1996
3. Fried & Lowry. "BigDog: Hierarchical Authentication, Session Control, and Authorization for the Web". USENIX Second Workshop on Electronic Commerce, Nov. 1996.
4. Gasser et al. "The Digital Distributed System Security Architecture", Proc. 1989 National Computer Security Conf., (1989), pp. 305-319
5. Gong et al. "Inside Java 2 Platform Security, Second Edition". Addison-Wesley (May 2003).

6. Lampson et al. "Authentication in Distributed Systems: Theory and Practice". ACM Trans. Comp. Sys., 10(4):265-310, Nov. 1992
7. Swift et al. "Improving the granularity of access control for Windows 2000". ACM Trans. Info. and Sys. Security, 5(4): 398-437, Nov. 2002.
8. Wallach et al. "SAFKASI: a security mechanism for language-based systems". ACM Trans. Soft. Eng. and Meth., 9(4): 341-378, Oct. 2000.
9. Wobber et al. Authentication in the Taos Operating System. ACM Trans. Comp. Sys., 12(1): 3-32, Feb. 1994.

PRESTO: A Predictive Storage Architecture for Sensor Networks*

Peter Desnoyers, Deepak Ganesan, Huan Li, Ming Li, Prashant Shenoy
University of Massachusetts Amherst

Abstract

We describe PRESTO, a predictive storage architecture for emerging large-scale, hierarchical sensor networks. In contrast to existing techniques, PRESTO is a proxy-centric architecture, where tethered proxies balance the need for interactive querying from users with the energy optimization needs of the remote sensors. The main novelty in this work lies in extensive use of predictive techniques that are a natural fit to the correlated behavior of the physical world. PRESTO exploits technology trends in storage to build an architecture that emphasizes archival at remote sensors and intelligent caching at proxies. The system also addresses user needs for querying such sensor networks by exposing a unified, easy to use data abstraction across numerous proxies and remote sensors.

1 Introduction

Many different kinds of networked data-centric sensor systems have emerged in recent years. Sensors generate data that must be processed, filtered, interpreted, cached, and archived in order to provide a useful infrastructure for users. Sensors are often untethered, and their energy resources need to be optimized to ensure long lifetime [2, 13]. Thus, energy-efficient data management is a key problem in sensor applications.

There are two commonly used models for processing data in sensor networks. The first model involves viewing the sensor network as a database [1, 2, 3], where queries are pushed all the way to the remote sensors. Such direct querying of the remote sensor nodes is generally more efficient energy-wise, since query-specific data processing can be performed at the data source to reduce communication requirements. However, such querying renders the system unusable for interactive use due to the high latency, low availability, and low reliability [4] inherent in duty-cycled, energy-limited wireless sensor networks. The second model has been one of data streams, where potentially useful sensor data is pushed from the sensors, and stored at a high-end server running a database. The database engine can perform statistical modeling and cleaning on the data [5], and provide lower latency, better availability, and better interactivity to user queries. However, this model is less energy efficient since it does not exploit the fact that only a subset of sensor data may be actually queried.

While both these models are important for current and future sensor networks, they have certain drawbacks. In this paper, we present PRESTO, a predictive store for sensor networks that attempts to provide the interactivity of the data streaming approach with the energy efficiency of the direct sensor querying. PRESTO differs from past work on data-centric sensor networks in several key respects (see Table 1).

Hierarchical Systems: Rather than designing our system for a single flat sensor network architecture, PRESTO reflects our philosophy that scalable sensor networks of the future will have multiple tiers, with a several tens of untethered sensors per tethered sensor proxy and several tens of sensor proxies per application. Being tethered, sensor proxies can be expected to be less resource constrained than the remote sensors, an aspect that PRESTO exploits in two different ways. Proxies cache current and past data from remote sensors and use predictive techniques on cached data to answer queries, thereby providing response times that are close to the data streaming approach. Proxies also use their superior processing capabilities to model, predict, and match query parameters to data dissemination at remote sensors, thereby providing the energy efficiency of the direct querying method.

Archival Queries: Unlike many systems that only support queries on the current sensor data [5], PRESTO supports archival queries on data that may be deemed to be interesting *post-facto*. The ability to query historical data is important in many sensor applications such as surveillance, where the ability to retroactively "go back" is necessary to determine, for instance, how an intruder broke into a building. Similarly, archival sensor data is often useful to conduct postmortems of unexpected and unusual events to better understand them for the future. PRESTO enables such PAST queries by employing a distributed archival store at remote sensors that records past sensor data; thereby resulting in a significantly different architecture from stream-based systems.

Single Logical View of Data: A key goal of PRESTO is to provide a unified data abstraction of a single logical

*† This work was supported in part by National Science Foundation grants EEC-0313747, CNS-0325868, CNS-0219520, EIA-0098060 and CNS-0323597.

Table 1: Comparison of PRESTO to related efforts.

	NOW Queries	PAST Queries	Prediction	Data Abstraction	Energy-Aware	Hierarchical design
Diffusion[2]	Direct sensor querying	No archival	No	Single remote sensor	Yes	No
Cougar[1]	Direct sensor querying	No archival	No	Single remote sensor	Yes	No
TinyDB[6]/BBQ[5]	Proxy querying	Archival at proxy	Yes	Single proxy	Yes	No
Aurora/Medusa[7]	Proxy querying	Archival at server	No	Distributed stream	No	P2P
PRESTO	Proxy querying + sensor querying on cache miss	Caching at proxy + archival at sensor	Yes	Single logical view of distributed store	Yes	Yes

store across tens to hundreds of proxies and thousands of remote sensors that comprise a sensor application. Part of this abstraction is enabled by the sensor proxy that abstracts the user from the vagaries of the remote sensor tier including lossy and unreliable sensor nodes and spatial and temporal consistency issues in the sensor data. The second enabling system component is a distributed index structure that constructs a unified view of caches across geographically distributed sensor proxies.

The novelty of PRESTO lies in its predictive storage capabilities and active interactions between proxies and sensors. Unlike traditional storage systems that are passive, the PRESTO proxy employs an active cache that predicts data values that are yet to be fetched from remote sensors (and thus, yet to be written to the local cache). While predictive techniques are also used in BBQ [5], we differ in that PRESTO uses active interactions to handle the occasional rare events that are inherently unpredictable. The PRESTO proxy provides feedback to remote sensors that limits the communication overhead of the sensor to data that is deemed "unpredictable" at the proxy. Such a predictive push-based approach ensures that rare, unexpected events are never missed, which is important in many event-driven applications such as intruder detection. When cache misses occur at the proxy, PRESTO reverts to direct querying of data archives at remote sensors.

Several technology trends make such a predictive storage architecture both feasible and appealing. First, radio communication is generally considered to be quickly reaching fundamental energy barriers [8]. Hence, the commonly held view is that communication should be reduced and compensated by increased use of either computation (up to four orders of magnitude less expensive [8]) or storage (two orders of magnitude less expensive (comparing 1Gb Samsung NAND Flash and Chipcon 802.15.4 radio). Second, capacities of flash memories continue to rise as per Moore's Law, and their costs continue to plummet. Thus, it will soon be feasible to build cost-effective sensor nodes with more than a gigabyte of flash memory. PRESTO exploits the presence of a large local store to reduce communication by archiving data locally at remote sensors whenever possible. Finally, processing speeds continue to increase, with new energy-efficient technologies delivering more CPU cycles per watt. This enables us to put more capable processors on remote sensors as well as intermediate proxies and leverage the additional processing capacity for extrapolation, batching, and compression, all of which can reduce communication.

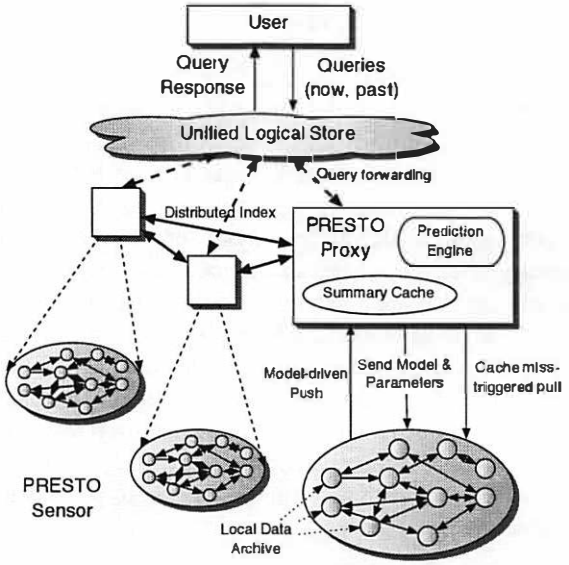


Figure 1: PRESTO architecture

The rest of this paper is structured as follows. Section 2 describes our system architecture. Sections 3, 4, and 5 describe the PRESTO proxy, sensor and the data abstraction, respectively. We conclude in Section 6.

2 System Architecture

Our view of the emerging sensor network architecture comprises three tiers as shown in Figure 1—a bottom tier of untethered remote sensor nodes, a middle tier of tethered sensor proxies, and an upper tier of user terminals.

The lowest tier is assumed to form a dense deployment of low-power sensors. A canonical sensor node at this tier is equipped with low-power sensors, a microcontroller, and a radio as well as a significant amount of flash memory (1GB). This tier may be heterogeneous, and might comprise different kinds of devices, sensors and platforms. In the future, some limited form of energy harvesting might assist these sensors in achieving

even greater lifetimes than is currently achievable. The common constraint for the lowest tier is energy, and the need for a long lifetime in-spite of it. The use of radio, processor, RAM, and the flash memory all consume energy, which needs to be limited.

The middle tier consists of power-rich sensor proxies that have significant computation, memory and storage resources and can use these resources continuously. Many different instances of this middle tier can be seen in different application settings. In urban environments, this tier would comprise a tethered base-station class node (e.g., Intel Stargate) with multiple radios—an 802.11 radio that connects it to a wireless mesh network and a low-power radio (e.g. 802.15.4) that connects it to the sensor nodes. Since Internet connectivity is widely available in many urban settings, these proxies may plug in to existing mesh networks or the wired infrastructure. In remote sensing applications [9], this tier could comprise a similar Stargate node with a solar power cell. Each proxy is assumed to manage several tens of lower-tier sensors in its vicinity. A typical sensor network deployment is will contain multiple geographically distributed proxies. For instance, if a building is being monitored, one sensor proxy might be placed per floor or hallway. At the highest tier of our infrastructure are users, who can query the sensor network through a query interface, perhaps using declarative queries as proposed in TinyDB [6].

System Operation: Although the PRESTO architecture does not preclude continual queries, in this paper, we focus on the mechanisms needed to support one-time queries on current and past sensor data. Each proxy is assumed to cache data summaries or a subset of the data from the lower tiers sensors. When a new query arrives, the proxy examines its cache to see if the data necessary to answer the query is available. In the event of a hit, the query can be processed locally. Cache misses are handled in one of two ways. The proxy first examines other cached data to see if the requested data can be extrapolated from it. Cached data from other nearby sensors or temporally adjacent data from the sensor can be used for such extrapolation. and the extrapolated data can be used to process the query locally. If the spatio-temporal extrapolation does not yields sufficiently accurate data to meet the query error tolerances, then the cache miss is handled by fetching data from other sensor caches or the archive at remote sensors. This is enabled by a complete local archive of past data at each remote sensor. On storage-constrained sensors, older archived data is aged gracefully to ensure that lower resolution representations are available [10].

To ensure that all "interesting" data is cached at a proxy with high probability, PRESTO employs a model-driven push approach. The prediction engine at the proxy builds models of correlations in the data and periodically transmits parameters of this model to the remote sensors. The remote sensors check their sensed data against this model and push data solely when the model fails, thereby saving energy-intensive communication at the expense of some cheaper computation. Such a model-driven push ensures that the proxy is notified of all significant drifts in sensor values as well as unusual changes caused by unexpected events. Observe that a pure pull-based approach can handle the former case but will likely fail to capture the latter scenario. In addition to an energy-efficient model-driven push, the PRESTO prediction engine also utilizes query characteristics such as query type, arrival rate, latency, and precision requirements to extract additional energy savings. For instance, sensors can be adaptively duty cycled and can employ batching to reduce their energy needs.

In the following sections, we describe the components of PRESTO in greater detail.

3 PRESTO Proxy

The PRESTO proxy comprises two components: a cache of summary information about the data observed at the remote sensors and a prediction engine that is responsible for data extrapolation, model-driven push, and query-sensor matching.

Sensor Data Cache: A central component of the sensor proxy is a summary cache of the data from remote sensors. This cache differs significantly from both memory caches as well as web caches in that the cached data is either a lossy view or a higher-level semantic event-based view of the sensor data. For instance, rather than sending the full data, sensors may transmit summaries of their observations to the proxy cache. Similarly, rather than sending raw data, a sensor may send processed events to the proxy. To illustrate, a camera sensor in a surveillance application may send notification that a new object has been detected and its type, rather than sending a raw image of the object. Such lossy or semantic representations of the data not only incur a smaller communication cost, they may be more appropriate from an application perspective. Further, the summary data cache at the proxy can be progressively refined as more accurate data is obtained from the remote sensors or as queries on past data results in missing portions of the cache being filled up.

Prediction Engine: The prediction engine at the proxy uses its prediction capabilities for three purposes: model-driven push, data extrapolation and query-sensor matching.

Model-Driven Push: PRESTO uses predictive modeling to enable model-driven push from the remote sensors. To do so, the proxy constructs a model that captures expected variations in the data and transmits pa-

rameters of this model to each remote sensor. For instance, a model of temperature variations will capture time-of-day effects (such as [5]) as well as the impact of seasons. Each remote sensor checks their sensed data against this model and transmits solely when the model fails, thereby saving energy-intensive communication at the expense of some cheaper computation. For instance, only deviations from the normal temperature for each hour of the day are reported. We seek a few important characteristics from these models. First, we require that models be *asymmetric*—they can be hard to build at the proxy, but they must require little resources to verify at the sensor. Thus, sensors must expend as little processing as possible to check if the sensed data conforms to the model. Second, the models should effectively capture the statistics of the underlying physical process corresponding to the sensor data. For instance, simple regression techniques and time-series analysis techniques may be used to model many temporal phenomena. Similarly, like in the acquisitional query processor work [5] a combination of multivariate models for the spatial axis and Markov model for the temporal axis can also be used for model weather data.

Extrapolation: The PRESTO prediction engine can also extrapolate missing data that are needed by a query. As explained earlier, extrapolated data can mask cache misses and answer queries so long as the query precision is met. Observe that the above predictive data models can serve the dual purpose of enabling data extrapolation at the proxy, while dictating which data needs to be pushed by the remote sensors. For instance, in the absence of failures and even when sensors do not report any observations, it is safe to assume that the temperature at a certain hour or the day conforms to the historical trends captured by the model. These values can be substituted for the actual observations and used to answer queries. Thus, data extrapolation enables the proxy to provide quick and accurate responses to queries even if the data corresponding to the query is missing from the cache. Our work builds on existing techniques such as multivariate data modeling proposed in TinyDB and BBQ [5].

Query-Sensor Matching: Finally, the PRESTO prediction engine is responsible for query-sensor matching to match the needs of queries to the operations of remote sensors. To maximize savings, sensors can be adaptively duty cycled and asked to batch and compress a set of data values prior to transmission. The proxy takes into account the characteristics of queries for such matching-based optimizations. The query type, frequency, latency and precision requirements are translated into the appropriate parameters for the remote sensors, such that they can minimize energy while achieving query requirements. For instance, if it is known that

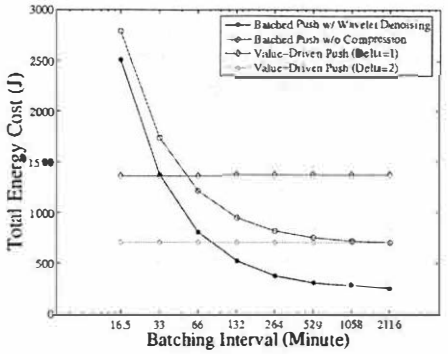


Figure 2: Exploiting batching to conserve energy

the worst case notification latency for typical queries is 10 minutes, the proxy can instruct remote sensors to set its radio duty-cycling parameters accordingly in order to conserve energy. The duty cycling parameters can be adaptively varied as new queries with different needs arrive into the system. Similarly, if the queries only require 75% precision in their response, lossy compression and aggregation techniques can be used to reduce the amount of transmitted data. The type of query can be exploited as well. For instance, scientists studying building health monitoring are typically interested in the mode of vibration of a building. The operation can be transmitted as a parameter to the sensor node, which uses the specified mode function on its local data before transmitting the final result.

Figure 2 shows one instance of such query-sensor matching in the case of temperature data [11], where the impact of batching on overall energy savings is demonstrated. Greater batching translates into two energy gains: (a) fewer packets imply a lower per-packet overhead including ACKs, packet headers and MAC-layer preambles, and (b) more batching results in better compression and data cleaning at the source of data, in this case, using wavelet denoising [12].

4 PRESTO Sensor

PRESTO is a proxy-centric architecture where much of the intelligence resides at the proxy, and the remote sensor is kept simple to enable efficient operation under resource constraints. Our contribution lies in the design of sensors that are simple, yet highly tunable and can be completely controlled by the proxy. The PRESTO sensor has two components. The first is an archival file-system that we are developing that provides energy-efficient archival of useful sensor data at each sensor as well as a simple time-based index structure to efficiently service read requests. Data archival at the remote sensors is needed to deal with queries on past data that may not be cached at the proxy. Such a data archive would not store all raw data, rather, it would only store sensor

data that is potentially useful for querying. For instance, in a traffic monitoring application, signatures of detected vehicles would constitute useful sensor data that is archived locally, whereas the sensor might use a classifier to process the sensor data and report the most likely vehicle type to the proxy. If storage is constrained on each sensor, graceful aging of archived data can be enabled using wavelet-based multi-resolution techniques [10]. The second component is a simple adaptive system that can use the information provided by the proxy to tune data transmission, data processing, aggregation, as well as duty-cycling parameters. For instance, in the case of a data collection query, lossy compression parameters (eg: using wavelets [10]) can be tuned by the proxy based on accuracy requirements of the queries.

5 PRESTO Data Abstraction

PRESTO aims to provide a single logical view of data that integrates archived data stored at numerous distributed remote sensors as well as caches and prediction models at numerous proxies. Such a view abstracts the user from variabilities at many levels—lossy and unreliable remote sensor network; spatial and temporal consistency issues in the sensor data; predictive responses from the proxy versus direct remote sensor querying; as well as bandwidth and connectivity issues in the case of wireless proxies.

The PRESTO data abstraction has three goals. The first is to provide different application-specific views of the distributed sensor data to enable efficient querying. For instance, a traffic monitoring network requires a view that preserves the order in which moving vehicles are detected across a spatial region. Such querying requires a single temporally ordered view of detections across distributed proxies and sensors. In our current work, we are exploring the use of order-preserving index structures such as Skip Graphs [14] for this purpose. The second goal of the data abstraction is dealing with temporal consistency issues that arise due to clock drift and skew across remote sensors as well as spatial consistency issues that arise due to overlapping coverage areas between proxies. Drift and skew of clocks at the remote sensors can result in erroneous timestamps, which need to be corrected to provide an accurate temporal view of data. In the spatial dimension, multiple proxies might be responsible for a group of sensor nodes for redundancy, reliability, and fault-tolerance reasons, and hence, cache consistency issues need to be addressed. Finally, the index structure will span a mix of wired sensor proxies with high bandwidth links and wireless 802.11-based proxies with lower bandwidth and availability. Hence, even if proxies cache data from remote sensors and provide predictive responses to queries, there might be variability in response times for queries due to the vagaries of 802.11 links. To deal with this problem, caches and prediction models at the wireless proxies may need to be further replicated at the wired proxies to enable low-latency query responses.

6 Discussion and Conclusions

We described PRESTO, a predictive storage architecture for emerging large-scale, hierarchical sensor networks. In contrast to existing techniques, PRESTO is a proxy-centric architecture, where tethered proxies balance the need for interactive querying from users with the energy optimization needs of the remote sensors. The main novelty in this work lies in extensive use of predictive techniques that are a natural fit to the correlated behavior of the physical world. PRESTO exploits technology trends in storage to build an architecture that emphasizes archival at remote sensors and intelligent caching at proxies. The system also addresses user needs for querying such sensor networks by exposing a unified, easy to use data abstraction across numerous proxies and remote sensors.

PRESTO can be used in different ways in different application contexts. Environmental weather patterns and commuter traffic patterns are examples of data that are highly predictable in the common case. PRESTO can enable the system to conserve energy by learning the predictable aspects of the data, and efficiently extracting only the unpredictable information from remote sensors. The unified data abstraction and predictive responses that PRESTO provides can be used in vehicle traffic querying as commuters can query the system to obtain quick responses. Surveillance applications can use the archival capability of PRESTO to query for event logs corresponding to past events. Activity monitoring applications such as elder care often involves a user wearing sensors that collect information about location, gait, posture as well as other daily activities [15]. PRESTO is particularly appropriate for such applications since daily activity patterns tend to be mostly predictable, with occasional unpredictable events or patterns that need to be explicitly reported to proxies.

While PRESTO has numerous interesting applications, there are multiple scenarios where this is not the right storage and querying model. Some applications might require extremely cheap sensors (eg: RFIDs) where the cost of augmenting each sensor with large local storage capacity may be prohibitive. Also, PRESTO may not be applicable in mission-critical applications where predictive responses can be misleading and have damaging consequences. While such applications will require different storage and querying architectures, we believe that PRESTO will have wide applicability across a range of data-intensive sensor network applications.

References

[1] P. Bonnet, J. Gehrke, and P. Seshadri, "Towards sensor database systems," in *Proceedings of ICMDM*, Hong Kong, January 2001.

[2] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: A scalable and robust communication paradigm for sensor networks," in *Proceedings of the ACM/IEEE Mobicom*, Boston, MA, USA, Aug. 2000.

[3] S. Ratnasamy, D. Estrin, R. Govindan, B. Karp, L. Yin S. Shenker, and F. Yu, "Data-centric storage in sensornets," in *ACM Hotnets*, 2001.

[4] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker, "Complex behavior at scale: An experimental study of low-power wireless sensor networks," Tech. Rep. UCLA/CSD-TR 02-0013, UCLA, Department of Computer Science, 2002.

[5] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong, "Model-driven data acquisition in sensor networks," in *VLDB*, 2004.

[6] S. Madden, M. Franklin, J. Hellerstein, and W. Hong, "Tag: a tiny aggregation service for ad-hoc sensor networks," in *OSDI*, Boston, MA, 2002.

[7] H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbets, and S. Zdonik, "Retrospective on Aurora," *VLDB Journal*, vol. 13, no. 4, 2004.

[8] G. Pottie and W. Kaiser, "Embedding the internet: wireless integrated network sensors," *CACM*, vol. 43, no. 5, pp. 51–58, May 2000.

[9] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil, and T. Schoellhammer, "A system for simulation, emulation, and deployment of heterogeneous sensor networks," in *Proceedings of ACM Sensys*, Baltimore, MD, 2004.

[10] D. Ganesan, B. Greenstein, D. Perelyubskiy, D. Estrin, and J. Heidemann, "Multi-resolution storage in sensor networks," in *Proceedings of ACM Sensys*, 2003.

[11] P. Bodik, W. Hong, C. Guestrin, S. Madden, M. Paskin, and R. Thibaux, ," Intel Lab Data. `http://db.lcs.mit.edu/labdata/labdata.html`.

[12] M. Vetterli and J. Kovacevic, *Wavelets and Subband coding*, Prentice Hall, New Jersey, 1995.

[13] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *Proceedings of ASPLOS-IX*, Cambridge, MA, USA, November 2000.

[14] G. Shah J. Aspnes, "Skip graphs," in *SODA*, Jan 2003.

[15] M. Philipose, K. P. Fishkin, M. Perkowitz, D. J. Patterson1, D. Hahnel, D. Fox, and H. Kautz, "Inferring ADLs from Interactions with Objects," in *IEEE Pervasive Computing*, volume 3, number 4, pp. 50-56, 2004.

Towards a Sensor Network Architecture: Lowering the Waistline

David Culler*, Prabal Dutta*, Cheng Tien Ee*, Rodrigo Fonseca*,
Jonathan Hui*, Philip Levis*, Joseph Polastre*, Scott Shenker*†,
Ion Stoica*, Gilman Tolle*, and Jerry Zhao†

1 Introduction

Wireless sensor networks have the potential to be tremendously beneficial to society. Embedded sensing will enable new scientific exploration, lead to better engineering, improve productivity, and enhance security. Research in sensor networks has made dramatic progress in the past decade, bringing these possibilities closer to reality. Hardware, particularly radio technology, is improving rapidly, leading to cheaper, faster, smaller, and longer-lasting nodes. Many systems challenges, such as robust multihop routing, effective power management, precise time synchronization, and efficient in-network query processing, have stable and compelling solutions. Several complete applications have been deployed that demonstrate all of these research accomplishments integrated into a coherent system, including some at relatively large scale [5, 17].

But the situation in sensornets, while promising, also has problems. The literature presents an alphabet soup of protocols and subsystems that make widely differing assumptions about the rest of the system and how its parts should interact. The extent to which these parts can be combined to build usable systems is quite limited. In order to produce running systems, research groups have produced vertically integrated designs in which their own set of components are specifically designed to work together, but are unable to interoperate with the work of others. This inherent incompatibility greatly reduces the synergy possible between research efforts and impedes progress.

It is the central tenet of this paper that the primary factor currently limiting research progress in sensornets today is not any specific technical challenge (though many remain, and deserve much further study) but is instead *the lack of an overall sensor network architecture.* Such an architecture would identify the essential services and their conceptual relationships. Such a decomposition would make it possible to compose components in a manner that promotes interoperability, transcends generations of technology, and allows innovation.

*CS Division, UC Berkeley, Berkeley, CA 94720.
†ICSI, 1947 Center Street, Berkeley, CA 94704

2 The Nature of an Architecture

At the highest level, an architecture decomposes a problem domain into a set of *services*, which are functional components, their mechanisms and their responsibilities. An architecture can also define a set of *interfaces* to its services, which are the structures and functions services expose their mechanisms with. Finally, at the lowest level, an architecture can specify its *protocols*, which include packet formats, communication exchanges, and state machines.

For interfaces and protocols, we say an architecture *can* define them because sometimes it is advantageous not to. For example, the Internet architecture precisely defines IP as a service (end-to-end communication, best-effort delivery, fragmentation/defragmentation, etc.) and as a protocol (packet format and semantics), but is ambivalent to the IP interface. Given the Internet's principal design goal — it is a *network* interconnection architecture — this ambivalence makes sense: IP does not want to dictate what software runs on each host.

In contrast to the Internet architecture, which seeks to promote communication interoperability, the POSIX architecture cares about software interoperability. Correspondingly, it cares greatly about interfaces while remaining ambivalent about the protocols. For example, the sockets service for end-to-end communication provides a precise interface, but has several underlying protocols (e.g., local communication, TCP, etc.).

The challenge in sensor networks is that their modes of operation introduce requirements and tradeoffs are very different from traditional systems. A sensor network application dictates sensor modalities, sample rates, real-time processing, data storage, and information exchange protocols among nodes. Early vision papers and analyses claimed that the traditional application/OS and network/data-link divisions are not well suited to sensor networks [3, 7], and community experiences building protocols and applications have shown this claim to be true [11, 16, 18, 27]. Thus, current sensor network software systems, such as TinyOS [9] relax these divisions and give developers the flexibility to define new ones. This relaxation has allowed researchers to re-examine core issues in scheduling, power-control,

and information flow by cutting across traditional service boundaries [14].

Cutting across boundaries, however, has led to monolithic solutions or to subsystem components with arbitrary interface assumptions. While research groups have each been able to build large and complex systems, the resulting services, interfaces, and protocols are incompatible with each other. For future work to be able to build on the prior efforts of others, we need a sensor network architecture, which will re-establish a meaningful separation of concerns.

The Internet architecture demonstrated how a properly chosen set of guiding principles and services can shape the evolution of a complex system over vast changes in technology, scale, and usage [2]. The philosophy of designing for heterogeneity, change and uncertainty was a radical shift from classical systems design, which more traditionally seeks a near optimal assembly of near optimal parts. Faced with integrating several existing networks with widely varying characteristics, the end-to-end principle and focus on interoperability led to a design that has successfully coped with tremendous growth and change. However, this design is not free of costs; the use of rigid layering sacrifices efficiency in various regards in return for increased interoperability.

The power of the Internet is revealed not so much in the elegance or efficiency of its individual services, but in its overall ability to adapt. This is one of our goals for developing an architecture for sensornets. We must be extremely mindful of any loss of efficiency for particular tasks as we seek to greatly enhance the interoperability between components and ability to advance.

The experiences and efforts of the sensor network community over the past years has helped discover exactly how the requirements and concerns of a sensor network architecture are different from the Internet, and how they are the same. The challenge in defining a sensor network architecture is deciding what to specify in its services and what to leave open. Specifying too little will force systems to re-implement functionality they cannot depend on, while specifying too much will constrain future technologies and possibly lead developers to discard the architecture. For this reason, we expect developing an architecture to be at first a growing and organic process. While conclusions from community experience have clearly converged on some issues, such as packet timestamps, others, such as aggregation, are still under debate. By starting with services (or even parts of services) for which there is consensus, an architecture will help focus the research debate on open problems, promoting forward progress.

3 The Narrow Waist

A complete sensornet architecture will need to address a family of specific issues, such as discovery, topology management, naming, routing and so on, but the overriding question is whether there is a "narrow waist" — a functional component representing a common service that permits a wide variety of uses above and a range of implementations below. At what level should it occur and what should it express? By requiring all network technologies to support IP, and all applications to run on top of IP, the Internet accommodates, even encourages, a vast degree of heterogeneity and diversity in both applications and underlying technologies. We have an analogous goal for sensornets; in both the application and device arenas we are in the midst of extremely rapid developments. Sensornets will only flourish if we can identify a narrow waist in the architecture that will allow devices and protocols to evolve and change without hampering optimization. The Internet has shown that the most important service of a network architecture is its narrow waist.

We claim that sensor networks can also have a narrow waist, the Sensor-net Protocol (SP). Unlike IP, which is a multihop protocol intended for end hosts communicating over a shared routing infrastructure, SP is a single hop protocol. The reason for this difference is simple: sensor networks use a wide range of multihop protocols, such as dissemination [15], flooding, tree routing [26], and aggregation [18]. Applications differ dramatically in their communication patterns and are intimately tied to their associated network protocols. Most applications neither require nor benefit from a common, universally routable addressing scheme. Those that do can build such protocols on top of SP.

The first step in developing our architecture is defining the SP service by deciding which mechanisms and functionality it provides and which it does not. Using SP, protocol designers must be able to design a range of efficient routing protocols independently of the underlying link layer. SP must facilitate in-network processing and collective communication as well as point-to-point transport. Moving the point of universal abstraction downward presents new issues that we do not typically concern ourselves about in the Internet architecture. It also requires a careful design of the layers above SP to provide a reasonably general platform on which to build various sensor network applications efficiently. If SP is to be a well defined service on top of a range of physical layers, how functionality divides across the packet boundary is a key question. To support the network protocols found in the sensornet literature, the mechanisms which a sender should be able to control include link level acknowledgments, post-media arbitration timestamping, retransmission and power manage-

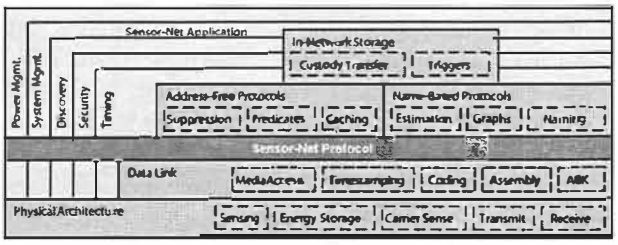


Figure 1: Sensor Network Service Decomposition

ment (cf. [20]). In addition to providing control points downwards, SP needs to expose costs upwards to higher layers so protocols can receive feedback on how to optimize their behavior as they exercise the available control.

For example, it is clear from community experience that the SP service must provide packet timestamps. Time synchronization research [6, 13] has shown that obtaining high precision timestamps on packet transmission and reception is inexpensive and can enable a wide range of synchronization algorithms above. While the need for this information is clear, exactly how it manifests is less so. There is consensus that when a node receives a packet, the SP service must provide a receive timestamp. As this timestamp is not a field of the packet that is received over the air, it is part of the SP service but not the protocol. The point of debate is on transmission. ETA [13] argues that transmitted packets should contain the sender's timestamp, while RBS [6] argues that this is unnecessary, as only the transmitter needs to know its timestamp. While both agree timestamps must be part of the SP service, ETA requires transmit timestamps to be part of the SP protocol, while RBS does not.

SP sits below many multihop protocols. Allowing higher level protocols to share control over an underlying communication medium raises concern as to how these protocols work together and cooperate. This is just the kind of investigation that the existence of SP would promote. We suggest that this question is tractable and very interesting in sensornets because they typically host a small number of widely distributed applications. In the Internet, such control is problematic because the infrastructure is shared by arbitrary applications anywhere in the world. The application specific nature of sensornets is more conducive to cross-layer and cross-application customization.

Therefore, rather than immediately specify protocols, our development of a sensor network architecture starts with defining SP as a service and providing a possible interface to that service so developers can test and evaluate it. Once, through literature analysis, communication with the community, and, of course, trial and error, we determine the boundaries of SP as a service, we can then focus on building and evaluating different candidate SP interfaces and SP protocols. We have begun this process by making a first attempt at defining SP as a service [21]. Trying to define a common service on top of very different underlying link layers (e.g., TDMA and CSMA) raises interesting questions about networking in this regime and suggests places where well-established networking terminology is ill-suited.

4 Filling In the Architecture

SP is the keystone of our sensor network architecture, bridging higher level protocols and applications to underlying data link and physical layers. Defining the SP service requires understanding the requirements of applications that lie above it and the capabilities of the technologies that lie below. Just as with IP, it is unlikely that SP will be ideally suited to all of its possible uses. However, by examining applications and their requirements, we can make educated decisions on what tradeoffs SP makes between its above and below pressures.

Applications today use a wide range of service layers, some of which have no clear analogues in the OSI model. For example, several commonly used communication services, such as collection routing [26] and dissemination [15], are *address-free*, in that, from the perspective from an application, there is no explicit destination. Of course, there are also name-based communication services, but the form and semantics of the naming are very different than end-to-end communication.

Address-free and name-based communication represent traditional service layers, which encapsulate underlying functionality. Our sensor network architecture also has *cross-layer services*, which cut across SP and indeed the entire architecture. Deployed applications have demonstrated that there are pieces of information and functionality which many different services require concurrently. Establishing the concept of cross-layer services allows existing approaches to continue while providing the structure necessary to promote composability and reuse.

4.1 Proposed Decomposition

Figure 1 shows a possible decomposition of a sensor network architecture. SP is the unifying service that bridges protocols and applications to the underlying data link and physical layers. Situated above SP are multiple network layer services, with applications selecting specific ones that suite the networking needs of the application. In Section 4.2, we discuss two of these upper layer services, name-based and address-free protocols. Situated below SP are underlying data link and physical layers, such as 802.15.4 [25] or S-MAC [28]. The diversity of functionality underlying layers present poses a variety of technical challenges to SP's design, which we discuss

and address in our SP proposal [21].

In addition to the layered services above and below SP, the architecture has cross-layer services, which Figure 1 shows on the left side. Cross-layer services include power management, timestamping/time synchronization, and discovery. As we discuss further in Section 4.3, these services are cross-layer in that they have uses across the entire spectrum of service layers.

This decomposition is far from complete. As sensor networks evolve and spread into new application domains, it is inevitable that new services will emerge. Current and foreseen future uses motivate our current decomposition, but it is also intended to be flexible enough to engender growth.

4.2 Address-free and Name-based

Unlike an IP network, which supports a single network addressing scheme and largely provides a single communication abstraction (*i.e.,* unicast), applications developed so far use a variety of naming schemes and multihop communication services. For example, the Line in the Sand event detection application routed along a 2D grid [5], the Great Duck Island habitat monitoring application routed up a collection tree [17] and Pursuer Evader Game tracking application used a landmark routing overlay on top of a tree-building algorithm [23].

This variety in naming and multihop communication is one of the main reasons behind our decision to push the narrow waist below the OSI network layer. Lowering the narrow waist allows the architecture to express and encompass this diversity both in the present and in the future. Trading off between the requirements of higher level services and the desire to keep SP as simple as possible is the principal first challenge in developing the architecture. In the remainder of this section, we describe two higher level services and how they might influence SP. Key architectural issues that arise in designing these services include route discovery and maintenance, naming, and the packet forwarding rules.

The address-free service layer encompasses a wide range of protocols, including flooding, collection routing [26], dissemination [15], and aggregation [4]. Although these protocols may include names to refer to data items — such as sequence numbers or dispatch IDs — they do not identify nodes directly. For example, when an application wants to send a piece of data up a collection tree, it does not need to specify a destination because it is implicit: the node's parent in the tree. The underlying collection tree routing protocol may address the parent directly, but it encapsulates this naming and hides it from layers above.

Unlike collection routing, however, which typically names nodes at the SP level, broadcast and dissemination protocols rely on the implicit naming provided by local connectivity. This represents an interesting SP design consideration, as some underlying MAC layers (e.g., TDMA-based MACs that turn off the radio) may not by themselves provide an efficient local broadcast primitive. This tension between the requirements of layers above and the capabilities of layers below demonstrates some of the difficulties that designing SP presents.

The name-based service layer encompasses multihop communication based on destination identifiers. This includes approaches such as geographic routing [12] and logical coordinate routing [8, 19], as well as more abstract and flexible naming schemes such as directed diffusion, which use data identifiers [11]. Global network names are powerful enough to support content-based storage within the sensor network, but require any-to-any routing [22].

In addition to packet forwarding, a node along a path can inspect received data and make local decisions regarding a packet based on its contents, possibly transforming the data before forwarding it, or suppressing it completely. This in-network processing can reduce communication while keeping higher-level semantic requirements. For example, when collecting a MAX query, which returns the maximum value of some variable, nodes need only forward the highest value they receive and suppress all other values.

The key observation is that the services above SP support very different semantics than those found in the network layer services of the Internet and OSI specifications. In particular, sensornets are primarily concerned with dissemination, collection, aggregation, and gradient-directed services, whereas the Internet is principally concerned with end-to-end communication [1].

4.3 Cross-Layer Services

One novel aspect of our sensor network architecture is the concept of cross-layer services. These services cut across layers or arise within multiple layers. Instead of being fully encapsulated at one layer, only visible to the layers above and below, cross-layer services are accessible to all of the layers in the system. In this section, we use power management to motivate the need for cross-layer services in a sensor network architecture, describe some of the research challenges they pose, and present timestamping as one example of such a service.

Energy constraints are a defining characteristic of sensor networks. Traditionally, power aware networking has dealt with a single point in the stack in isolation. This approach is not practical in sensornets because power management often appears in many places and takes many forms. Below SP, power aware MACs attempt to turn off the radio invisibly to the stack above [24]. Within SP, buffering multiple packets and

sending them back-to-back in a burst can be more efficient than sending them individually as they appear from the network layer services [21]. Routing layers above SP can have multihop flow information that allows them to schedule future radio activity [10]. Applications can have their own scheduling policies: TinyDB shuts down the whole networking stack between query processing epochs [16].

As a consequence of its ubiquity, power management is particularly challenging to abstract into a clean architectural concept. The architecture must allow many different services from very different levels to collaborate and work together. These services must therefore be accessible to all levels of the system. On one hand, these services must have policies to arbitrate between conflicting requests; on the other, constraining the possible policies unnecessarily will hamper future growth. An architecture must establish clear guiding principles and sufficiently rich, yet loosely-coupled and appropriately abstracted, interfaces to support cross layer services.

All of the power management approaches mentioned above use some form of time synchronization to schedule communication. All of these time synchronization algorithms depend on having accurate packet timestamps. Therefore, these timestamps must be information that cuts through layers so sub-SP as well as super-SP services can use them. While time synchronization services can be situated above SP, MAC-layer timestamping below SP greatly improves their precision [13]. By choosing an to generate this data at an idealized point in the communication stack (e.g., post media arbitration) SP can achieve microsecond resolution inexpensively.

Timing information must cross layers so many services can take advantage of them. The sensor network architecture therefore provides it in cross-layer services. The preferred method of exposing timestamps in the link interface, and more generally across the architecture, is an important design point that must be addressed with an eye toward removing any temptation for time coordination services to circumvent SP. Power management is an example of a cross-layer service for downward control; timestamps are an example of a cross-layer service for upward information flow.

While this section presented only two cross layer abstractions, there are many more that we need to address. Examples include system management, discovery, and security. These services need to be accessible to all of the layers in the system so their abstractions present a central challenge to the architecture's design: developing a methodology for providing interfaces rich enough for application/system collaboration while remaining flexible enough to encompass growth and evolve as time rolls forward.

5 Conclusion

We contend that the main obstacle limiting progress in sensornet work is the lack of an architecture. A sensor network architecture would factor out the key services required by applications and compose them in a coherent structure, while allowing innovative technologies and applications to evolve independently. We argue that the narrow waist of this architecture should not be a network layer as in the current Internet, but single-hop communication with a rich enough interface to allow a diverse range of network protocols. This design decision is driven by the fact that, unlike an IP network, sensornets require a wide variety of naming schemes and multihop communication services.

However, there are many questions that need to be answered before such an architecture becomes a reality. Chief among those are the functionality provided by the SP service, the functional decomposition of sensor networking into services now that the narrow waist is single hop, and how cross-layers services such as timestamping can be designed to enable a broad spectrum of uses while minimizing complexity. Our hope and goal is that such an architecture will enable research groups to more easily collaborate and build on each other's efforts. Rather than a set of incompatible and vertically integrated systems, we will in the near future see in sensor networks the variety and innovation we see in the Internet today.

Acknowledgements

This work was supported by the National Science Foundation, under grant CNS-043545.

References

[1] B. Carpenter. Architectural principles of the internet. Request For Comments 1958, June 1996.

[2] D. D. Clark. The design philosophy of the DARPA internet protocols. In *Proceedings of the SIGCOMM '88 Conference.*

[3] N. R. C. Committee on Networked Systems of Embedded Computers. *Embedded, Everywhere: A Research Agenda for Networked Systems of Embedded Computers.* National Academy Press, Washington, DC, USA, 2001.

[4] J. Considine, F. Li, G. Kollios, and J. Byers. Approximate aggregation techniques for sensor databases. In *Proceedings of the 20th International Conference on Data Engineering (ICDE '04).*

[5] P. Dutta, M. Grimmer, A. Arora, S. Bibyk, and D. Culler. Design of a wireless sensor network platform for detecting rare, random, and ephemeral events. In *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks (IPSN '05).*

[6] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. In *Proceedings Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002).*

[7] D. Estrin, R. Govindan, J. S. Heidemann, and S. Kumar. Next century challenges: Scalable coordination in sensor networks. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '99)*.

[8] R. Fonseca, S. Ratnasamy, J. Zhao, , C. T. Ee, D. Culler, S. Shenker, and I. Stoica. Beacon vector routing: Scalable point-to-point routing in wireless sensornets. In *Proceedings of the Second USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI 2005)*.

[9] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000. TinyOS is available at http://webs.cs.berkeley.edu.

[10] B. Hohlt, L. Doherty, and E. Brewer. Flexible power scheduling for sensor networks. In *Proceedings of the Third International Symposium on Information Processing in Sensor Networks*, Berkeley, CA, Apr. 2004.

[11] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking (MobiCom '00)*.

[12] B. Karp and H. T. Kung. GPSR: greedy perimeter stateless routing for wireless networks. In *Proceedings of the Sixth International Conference on Mobile Computing and Networking (MobiCom 2000)*.

[13] B. Kusy, P. Dutta, P. Levis, M. Maróti, Ákos Lédeczi, and D. Culler. Elapsed time on arrival: A simple, versatile, and scalable primitive for time synchronization services. *International Journal of Ad hoc and Ubiquitious Computing*, 2005.

[14] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The emergence of networking abstractions and techniques in tinyos. In *Proceedings of the First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2004.

[15] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code maintenance and propagation in wireless sensor networks. In *Proceedings of the First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2004.

[16] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proceedings of the Fifth ACM Symposium on Operating System Design and Implementation (OSDI 2002)*.

[17] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the ACM International Workshop on Wireless Sensor Networks and Applications*, 2002.

[18] S. Nath, P. B. Gibbons, Z. Anderson, and S. n Seshan. Synopsis diffusion for robust aggregation in sensor networks. In *Proceedings of the Second ACM Conference on Embedded Networked Se nsor Systems (SenSys)*, 2004.

[19] J. Newsome and D. Song. Gem: graph embedding for routing and data-centric storage in sensor networks without geographic information. In *Proceedings of the first international conference on Embedded networked sensor systems*. ACM Press, 2003.

[20] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *Proceedings of the 2nd ACM Conference on Embedded Network Sensor Systems*, 2004.

[21] J. Polastre, J. Hui, P. Levis, J. Zhao, D. Culler, S. Shenker, and I. Stoica. A unifying link abstraction for wireless sensor networks. In *Proceedings of the Third ACM Conference on Embedded Networked Sensor Systems (SenSys 2005)*.

[22] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT: a geographic hash table for data-centric storage. In *Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications*, 2002.

[23] C. Sharp, S. Schaffert, A. Woo, N. Sastry, C. Karlof, S. Sastry, and D. Culler. Design and implementation of a sensor network system for vehicle tracking and autonomous interception. In *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN)*, 2005.

[24] S. Singh and C. S. Raghavendra. Pamas: power aware multi-access protocol with signalling for ad hoc networks. *SIGCOMM Comput. Commun. Rev.*, 28(3):5–26, 1998.

[25] The Institute of Electrical and Electronics Engineers, Inc. Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs), Oct. 2003.

[26] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of the First International Conference on Embedded Networked Sensor Systems (SenSys)*, 2003.

[27] N. Xu, S. Rangwala, K. K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A wireless sensor network for structural monitoring. In *Proceedings of the 2nd international conference on Embedded networked sensor systems (SenSys)*, 2004.

[28] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient mac protocol for wireless sensor networks. In *Proceedings of the 21st International Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002)*, New York, NY, June 2002.

Green team paper: Falling off the cliff: when systems go nonlinear

Yvonne Coady, Russ Cox, John DeTreville, Peter Druschel, Joseph Hellerstein, Andrew Hume, Kimberly Keeton, Thu Nguyen, Christopher Small, Lex Stein, Andrew Warfield[1]

Abstract

As the systems we build become more complex, understanding and managing their behavior becomes more challenging. If the system's inputs are within an acceptable range, it will behave predictably. However, the system may "fall off the cliff" if input values are outside this range. This nonlinear behavior is undesirable, because the system no longer behaves predictably: it may not be possible to use, control or even recover the system. In this paper, we describe what it means for a system to fall off the cliff. We outline methods for detecting and predicting these modes of nonlinear behavior, and propose several approaches for designing systems to cope with these instabilities, or to avoid them altogether. We conclude by outlining open research questions for investigation by the systems community.

1. Introduction

A system behaves *nonlinearly* when the same input or environmental change produces different system behavior at light loads and at heavy loads. For instance, an increased demand at light loads might produce *linear* behavior, where the amount of work the system performs is proportional to the load, and the system quickly recovers from perturbations. Systems may operate nonlinearly under heavier loads, producing poor or unpredictable performance and perhaps even resulting in a partial or complete system collapse.

Systems respond to input changes like increased load in different ways. Some employ various techniques to reduce load and bring the system back to a "safe" mode of operation. Alternately, they may gracefully degrade performance or otherwise reduce the quality of service provided for each request. In the absence of such coping mechanisms, systems may degrade gracelessly, resulting in a loss of predictability, recoverability, or controllability.

The real world is full of systems that exhibit each of these strategies. One example is the US telephone network. To reduce capital costs, local telephone switches are configured to handle some limited number of concurrent calls that is far below the theoretical limit. As a result, an emergency that causes a sudden spike in call attempts can swamp local systems. Customers who do not receive service promptly often hang up and retry repeatedly, nonlinearly increasing the system load and lengthening the period during which they do not receive service.

The telephone network incorporates a number of effective techniques to handle overload more gracefully. For example, although call attempts are normally handled in FIFO order, during overload they are handled in LIFO order, increasing the fraction of customers who receive prompt service and thereby reducing retries. These techniques have evolved over a period of decades, producing a stable telephone system, but this lengthy evolutionary path is not available to more modern systems.

Another example of graceful degradation is the reaction of the CNN.com web service to increased demand after the terrorist attacks on September 11, 2001 [9]. In the span of fifteen minutes, page request demand increased by an order of magnitude: peak demand rose to 1.8 million hits per minute, which is 20X of normal demand. The organization employed several techniques to handle the load spikes. First, they dynamically reprovisioned servers from other web services (e.g., cartoons or entertainment) to reinforce their news service. They also added additional server capacity to the system. Most interestingly, they chose to reduce the com-

[1] Author affiliations and email addresses: University of Victoria, *ycoady@cs.uvic.ca*; MIT, *rsc@mit.edu*; Microsoft Research, *johndetr@microsoft.com*; Rice University, *druschel@cs.rice.edu*; IBM Research, *hellers@us.ibm.com*; AT&T Research, *andrew@research.att.com*; HP Labs, *kimberly.keeton@hp.com*; Rutgers University, *tdnguyen@cs.rutgers.edu*; Vanu, Inc., *chris@vanu.com*; Harvard University, *stein@eecs.harvard.edu*; University of Cambridge, *andrew.warfield@cl.cam.ac.uk*

plexity of the pages they serviced, by removing advertisements and pictures, and focusing on the text.

The real world also provides examples of graceless degradation. On August 14, 2003, a series of operator errors caused an electrical power grid in northern Ohio to stop tracking failures caused by sagging wires on a hot day [2]. Power lines sag as they carry power, and sometimes fail after contacting trees, forcing reallocation of power through other wires, and making them fail, too. Later, as the outage spread, overload sensors in other areas produced false positives and shut down more of the system, eventually affecting 50 million people in the United States and Canada. Power was not fully restored in some areas for over a week.

We would like our systems to behave more like the telephone network or CNN.com, rather than the power grid on that hot August day. Can we design and build systems whose behavior we can understand and that do not behave unexpectedly under unusual circumstances? Can we build systems that work all of the time, not just much of the time? Or are complex systems inherently unpredictable under unusual loads?

2. Defining graceless degradation

Graceless degradation describes the situation in which a small change in a system's input or environment causes a large degradation in system behavior. We take a broad view of change, including an increase in load, a modification of configuration, or the installation or upgrade of a system component. Similarly, we define degradation broadly to include a loss of *predictability*, *recoverability*, and *controllability*. This section characterizes these types of degradation and discusses why they occur.

Types of graceless degradation

Probably the most familiar form of degradation is the loss of predictable performance for a service. For instance, in the management of virtual memory, a small increase in the multiprogramming level can result in highly variable response times if not all working sets can fit into memory. Alternately, in the configuration of database management systems, a small increase in buffer pool size can rapidly degrade throughput if it results in a change in query plans. In these examples, a small change in concurrency level, load, or data volume results in a tremendous change in system performance.

A common cause for a loss of predictable service under load is the use of load adaptation mechanisms that introduce feedback loops into the system. An example of such feedback is *thrashing* – the situation in which a virtual memory system is so constrained in satisfying the physical memory requirements of a set of concurrent applications that it spends the majority of its time moving pages of memory to and from disk rather than making forward progress. Still another example is TCP's congestion control mechanism. TCP interprets packet loss as an indication of congestion and halves a connection's transmission rate in response; this behavior results in poor performance on even moderately lossy links [4]. This congestive response has been shown to be vulnerable to attacks on TCP flows, especially where TCP implementations use constant retry timeouts; well-timed bursts of data have been shown to render the TCP connections on a link useless [7, 8].

A second type of graceless degradation is the loss of recoverability of the critical resources being used or provided by a system. In a storage system, the data being stored is a critical resource. As the underlying storage system degrades (e.g., as physical disks fail), this data is itself in a degraded state: it is less capable of surviving further failures, and may not be provided at as high a throughput as in the fully-functional system. After sufficient degradation, the resources are irrecoverably lost. This form of degradation results in a compromise of the system's capacity to provide its service.

A final type of graceless degradation is a loss of controllability. Often, as a system degrades, so does the ability to intervene to prevent further decline. A naïve example is that of a UNIX system experiencing a *fork-bomb*, in which a malicious process alternates between consuming system resources and forking copies of itself. An administrator wanting to recover the system needs to kill the forking processes, but as time progresses must kill more and more processes, with less and less resources available to recover.

Why does graceless degradation happen?

As system designers, we hope to build systems that are stable and predictable under all possible (or at least all specified) operating conditions. We now consider a set of specific characteristics of existing systems that lead to graceless degradation. This list is hardly comprehensive, but rather an attempt to identify some key factors of concern.

Renewable resource exhaustion: systems that allow over-subscription of renewable resources (CPU, memory, and network connections) are susceptible to over-

load. While overloaded, the system will have to provide some means of dividing the limited resources across consumers until the load returns to an acceptable state. Although overloading of renewable resources is not necessarily a cause of graceless degradation, the mechanisms for dealing with it frequently are.

Persistent resource exhaustion: As discussed with respect to the loss of recoverability above, the persistent resources of a system may themselves be compromised. This situation may occur due to failure of the underlying devices, system or application software, operator mistakes, or malicious attacks. Systems may be designed to be resistant against this form of degradation by employing various forms of redundancy.

Feedback-induced degeneration: Adaptation mechanisms within a system that feedback into the system's operational behavior may enter states of oscillation or related instability, and thus prevent the system from getting useful work done.

Removal from expected operating regime: A superset of the previous example, a system may be forced into an unexpected mode of operation. Such a phase change may result in the execution of poorly-tested code paths and compromise the stability of the system as a whole.

Degradation of operating state over time: Small, non-performance-critical problems may accumulate over the course of a system's lifetime. These state permutations may result in difficulties much later. Consider the management of applications on modern operating systems, in which *OS rot* may eventually result in the inability to update a system, requiring that the OS be re-installed from scratch.

Error conditions or exception logic: Exception logic is rarely invoked and often poorly tested. Thus, when it is invoked, it is common for degradation or even failures to result. Often exception logic is invoked as a result of a small increment in load that causes a buffer pool to overflow or too many file handles to be acquired. Thus, while the change in workload appears to be small, the change in the execution path is substantial.

Unintended software reuse: Modular software design encourages the reuse of components, as well as the construction of hierarchical systems from existing components. Although this approach can reduce costs, it can also lead to successful systems being used in ways and environments never imagined. Components can behave predictably in their intended environment, but unpredictably in others.

Unclear usage semantics: The premise of this section is that a small change results in a large degradation in performance. But sometimes it is difficult to quantify "small." For example, is it a small change to increase a buffer pool size by 1KB in a database management system with 1GB of memory? While this is small in terms of the fraction of memory affected, it may be huge in terms of the impact on query plans. To address this case and the previous one, it may be valuable to specify constraints on how software components can be safely used, and to verify the satisfaction of these constraints before deployment or during execution.

3. Detecting graceless degradation

Computer systems susceptible to graceless degradation should be built to detect such graceless degradation and substitute a more graceful alternative. The first step in such an approach is monitoring the system to detect when graceless degradation occurs or is about to occur.

If we view the system as a black box with inputs (e.g., request load, hardware failures) and outputs (e.g., throughput, latency, correctness, and other application-specific performance measurements), then a basic detection strategy is to characterize the safe operating ranges for the inputs or the outputs (or both) and detect when the system has moved outside the safe operating range.

Some operating constraints can be derived from the design of the system. For instance, a decision to use erasure codes places a hard limit on the number of fragments that must be available. Other constraints might be derived from more general requirements: a web server should respond to a request within a minute or else the response is likely to be ignored by the web browser – either the program or the human, both of which are likely to have timed out. The most precise constraints can only be derived from testing the system and determining what operating conditions keep it performing as desired.

Testing a large computer system may be non-trivial: the CNN.com web servers reached over one million page views per minute following both the 2000 U.S. elections and the 2001 terrorist attacks. Generating such conditions during testing requires a significant test framework. Computer systems designers might take solace in the fact that, unlike physical structures such as bridges, computer systems are usually not destroyed by being tested beyond their limits.

System load is not the only interesting parameter during testing. For example, testing a web server farm might also mean checking how many server failures the farm can tolerate simultaneously without entering a cascading failure scenario. Parameters are often interrelated: in the previous example, offered load certainly affects the number of server failures that can be tolerated.

Black-box testing may be insufficient for applications that are expected to run for long periods of time, as it is difficult to identify inputs and environmental conditions that can drive an application into unsafe regimes. Currently, some researchers are exploring the alternative *white-box* testing approach. For example, if source code is available (or byte code for Java applications), the compiler may be able to help. Compiler analyses can aid in the coverage testing of *uncommon code paths* such as recovery code [6]. Compiler analyses and instrumentations may also help applications and the runtime system/OS to track resource usage and detect when an application may be approaching the cliff.

Testing should focus on determining safe output parameters, as well. For example, if a web server can respond to all requests within five seconds under expected conditions, then significantly longer response times in a real deployment are indicative of unexpected behavior, possibly graceless degradation. Safe operating ranges could also be defined in terms of more cumulative statistics. For example, high variance in one-minute average throughput during high load might indicate that the servers are experiencing performance problems.

Once the expected safe operating parameters have been determined, the system must be able to continuously measure and check these parameters, ready to change behavior if graceless degradation is detected or anticipated. Statistical learning techniques may provide a means for understanding the observed data [5].

4. Coping with graceless degradation

There are several ways to cope with and even avoid graceless degradation. Admission control limits the amount of load that can enter a system. Overprovisioning builds a buffer of extra resources into a system. Reprovisioning dynamically adds resources as needed. Load shedding drops or scales back processing when resource over-commitment is detected.

Admission control conditions system load to try to avoid load spikes. Unlike physical systems, which often have implicit capacity-based admission controls, computer systems cannot depend on physical space or fixed environmental conditions to impose limits. As a result, computer systems must explicitly control admission. Examples of admission control in computer systems include circuit signaling in computer networks or user login. However, many computer systems (e.g., IP networks or web servers) use very little admission control. Admission control and overprovisioning are duals. An ideal admission control scheme conditions load so that it can never take a system out of its safe region. Overprovisioning makes the safe region so vast that the cliff is over the horizon.

Computer systems tend to underprovision for efficiency, rather than overprovision for safety. When compared with bridges, buildings, or other physical systems, most computer systems are designed with few excess resources. Overprovisioning presents two challenges. First, it is expensive. Second, it is difficult to know how much to overprovision each resource. Overprovisioning to handle load spikes smoothly means that most resources will be idle most of the time. Statistical multiplexing increases resource utilization by gambling on uncorrelated load. When the requests become correlated, the system receives a burst, and the gamble has been lost. At this point, the system is approaching the cliff and has to choose a strategy for coping. It can try to reprovision resources to move the cliff farther away, or it can use short-term approaches, such as load shedding, to back away from the cliff.

While software complexity may make it difficult to define a safe region *a priori*, the flexibility of software control provides a means to rescue systems that are leaving their safe region and heading for the cliff. Software can reprovision and reorganize system resources in real-time. For example, many storage systems use a virtualization layer to make capacity addition and failure events transparent to applications. In contrast to overprovisioning, where resources are statically allocated to absorb peak load, reprovisioning either changes the mix of resources or includes resources from an external source. For example, resources could be incorporated from a pool shared across many systems. In this case, reprovisioning is an attempt to statistically multiplex overprovisioning across independent systems. Systems that share a resource pool should have uncorrelated needs for the pool to remain solvent.

Reprovisioning is particularly important for situations where falling off the cliff implies loss of recoverability. For example, consider data redundancy for availability and durability. If the data is replicated using erasure coding, when the number of fragments drops below a

critical threshold, then the data becomes unrecoverable. Consider an erasure code-based P2P storage system. If replicas are failing and some data are approaching their critical threshold, then it becomes necessary to reprovision storage nodes, even at the expense of handling incoming load. Incoming load could be throttled down through admission control or the load shedding approach discussed below. This example illustrates that coping mechanisms can be combined effectively. TotalRecall is an example of such a system: it automatically measures and estimates the availability of host components and calculates and enforces the appropriate redundancy mechanisms and repair policies [1].

A final approach for avoiding graceless degradation is load-shedding. The simplest approach to shedding load is to drop requests from the tail of a FIFO queue. Another approach is to prioritize and postpone work where possible. One example of this approach is soft updates, which stabilize file system performance under heavy load by tracking the dependencies between block I/Os to postpone disk updates until the system calms. In the extreme case of the load shedding approach, a system might choose to avoid a cliff by resetting its state and starting fresh through either a full or partial reboot [3].

Load shedding may provoke feedback from the higher-level systems that issued the dropped requests. If the feedback is poorly behaved, it threatens to further aggravate an already struggling system: consider the phone retry example from Section 1. Synchronization is a danger because it correlates load and neutralizes statistical multiplexing. Randomization can help avoid synchronization. Exponential backoff can also reduce feedback problems by progressively delaying consistently problematic retries [10]. Negative acknowledgments (nacks) avoid generating load on an overloaded system by using acks for success cases and not sending any reply for errors, letting the higher layer time out. These approaches have their limits, however: they assume that the upper layer is a trusted and logical system, which may not always be the case.

Many systems are designed under the assumption of particular environmental parameters. Using randomization is one way to immunize a system against fluctuations in these parameters. The system will not behave optimally under some conditions, but at least it will not perform terribly under others. Randomized file system layout has been shown to provide stable file system performance across storage system virtualization parameters [11]. For routing in a hypercube, sending first to a random neighbor has been shown to improve performance by balancing messages across queues [12].

5. Summary and open research questions

Catastrophic failures have forced us to consider what should be done to better understand and manage system software. Avoidance and detection strategies require that we not only clearly define where the cliffs are, but also identify trends that force systems towards them. Key future challenges thus revolve around identifying a meaningful set of system constraints to describe safe operating regions, effectively capturing information about the system's operational state, and responding to cliff-inducing conditions in a timely fashion.

System constraints must be holistic to be meaningful. Some set of local system constraints may be known *a priori*, while potentially global constraints must be dynamically derived from specifics of system configuration and execution environment. Open questions include how best to identify and represent constraints or safe modes of operation, how to expose the right parameters for local constraints and how to dynamically derive context-specific holistic constraints.

Testing the system may help to discover operating constraints. Required advancements in this area include trace collection of heavy load scenarios, workload generators to synthetically generate load or to replay collected traces, and development of large-scale simulation and/or emulation environments.

The process of capturing and mining system state introduces several challenges. Given the vast amount of shared system state and increasing variability of configuration options, research challenges include how to manage state collection carefully, how to selectively monitor state according to global/local information needs, and how to quantify critical tradeoffs in safety and performance.

Responding to potentially cliff-inducing conditions requires an appropriate coping strategy. Further research is required to define new approaches for enforcing safe modes of operation and for gracefully degrading system behavior, and to understand the conditions under which each strategy may be appropriate.

Given the nature of this problem and the dramatic increase in its importance, we as a research community must collectively commit to better understanding and managing the systems we build.

6. References

[1] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. M. Voelker. "TotalRecall: systems support for automated availability management," *Proc. of ACM/USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, March 2004.

[2] Canada-U.S. Power System Outage Task Force. *Final report on the August 14, 2003 blackout in the United States and Canada: causes and recommendations*. April 2004. Available from http://www.nrcan-rncan.gc.ca/media/docs/final/finalrep_e.htm.

[3] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot - a technique for cheap recovery," *Proc. of 6th ACM/USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, December 2004.

[4] R. Chakravorty, J. Cartwright, and I. Pratt. "Practical experience with TCP over GPRS," *Proc. of IEEE GLOBECOM*, Taipei, Taiwan, November 2002.

[5] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. Chase. "Correlating instrumentation data to system states: a building block for automated diagnosis and control," *Proc. of 6th OSDI*, San Francisco, CA, December 2004.

[6] C. Fu, B. Ryder, A. Milanova, and D. Wonnacott. "Robustness testing of Java server applications," *IEEE Trans. on Software Engineering*, Vol. 31, No. 4, April 2005.

[7] M. Guirguis, A. Bestavros, and I. Matta. "Exploiting the transients of adaptation for RoQ attacks on Internet resources," *Proc. of Intl. Conf. on Network Protocols*, Berlin, Germany, October 2004.

[8] A. Kuzmanovic and E. Knightly. "Low-rate TCP-targeted denial of service attacks. (The shrew vs. the mice and elephants)," *Proc. of ACM SIGCOMM*, Karlsruhe, Germany, August 2003.

[9] W. Lefebvre. "CNN.com: facing a world crisis." Invited talk at *USENIX 15th Systems Administration Conference (LISA)*, San Diego, CA, December 2001. Summary available from http://www.usenix.org/publications/library/proceedings/lisa2001/lisa2001confrpts.pdf.

[10] R. Metcalfe and D. Boggs, "Ethernet: distributed packet switching for local computer networks", *Communications of the ACM*, Vol. 19, No. 5, July 1976, pp. 395 - 404.

[11] L. Stein. "Stupid file systems are better," *Proc. of ACM/USENIX 10th Workshop on Hot Topics in Operating Systems (HotOS X)*, Santa Fe, NM, June 2005.

[12] L. Valiant. "A scheme for fast parallel communication," *SIAM Journal on Computing*, Vol. 11, No. 2, 1982, pp. 350 - 361.

The Many Faces of Systems Research – And How to Evaluate Them

Aaron B. Brown
IBM Research

Anupam Chanda
Rice University

Rik Farrow
Consultant

Alexandra Fedorova
Harvard University

Petros Maniatis
Intel Research

Michael L. Scott
University of Rochester

Abstract

Improper evaluation of systems papers may result in the loss or delay in publication of possibly important research. This paper posits that systems research papers may be evaluated in one or more of the three dimensions of science, engineering and art. Examples of these dimensions are provided, and methods for evaluating papers based on these dimensions are suggested. In the dimension of science, papers can be judged by how well they actually follow the scientific method, and by the inclusion of proofs or statistical measures of the significance of results. In engineering, the applicability and utility of the research in solving real world problems is the main metric. Finally, we argue that art be considered as a paper category evaluated based on elegance, simplicity, and beauty.

1 Introduction

Evaluating systems research[1] is hard. Systems research is multifaceted; it often involves proving scientific hypotheses as well as designing and implementing real systems. As such, it goes beyond traditional science, spreading into the realm of engineering and perhaps even art, as system designers strive for elegance in their systems' blueprints. In this paper, we argue that evaluation criteria for systems research should match the dimension — engineering, science, art — in which particular work extends.

Every systems project maps to different points on each of these dimensions. A study evaluating performance of a new system could be regarded as "engineering" research. However, the ultimate goal of a performance study is typically to prove or disprove a hypothesis, to find the "truth," and this manifests the scientific dimension of the study. Likewise, research that might at first be classified as "art" can sometimes contribute to "science" or "engineering" as well, particularly if it introduces a new perspective that simplifies scientific understanding or improves ease of use. Failure to recognize the multidimensionality of systems research leads to subjective evaluation and misuse of evaluation criteria—for example engineering metrics are used to evaluate an "artistic" or a "scientific" research result.

In the rest of the paper we describe in more detail these research categories, partly prescriptively by specifying the qualities that exemplars of each dimension exhibit, and by example by pointing out specific instances of each (Section 2). In Section 3, we expand on the evaluation criteria that appear appropriate for each dimension, drawing from the non-CS disciplines after which each dimension is modelled. In Section 4 we propose an action plan for the improvement of systems research evaluation based on these three dimensions. Finally, we present related work in Section 5 and then conclude.

2 Dimensions of Systems Research

In this section, we identify the driving forces within the evaluation dimensions of science, engineering, and art. We also describe instances whose principal components exemplify individual or combined dimensions.

2.1 Science

Ironically, not many obvious opportunities exist readily in Computer Science to conduct Science. Computer artifacts are human-made; they are not natural parts of the physical world and, as such, have few laws to be discovered that computer engineers did not themselves instill into the field (although this perspective is slowly changing as systems grow so complex that they begin to exhibit emergent behavior). And yet, computer science deals with computers, which typically run with electrons carrying information bits, which in turn are governed by the laws of physics as much as any other aspect of the physical world. In the traditional terminology of databases, computer science works on a view of the ground truths of the universe, as manipulated by com-

puter architecture, by software, and by the applications that business, "mainstream" science, and entertainment have demanded. In this manner, science in Computer Science deals with studying the manifestation of the laws of the universe in the artifacts of the field. Consequently, "scientific" systems research strives to discover truth, by forming hypotheses and then proving or disproving those hypotheses mathematically or via experimentation, as well as identifying the effects and limitations of computer artifacts on the physical world.

A typical scientific example from the systems literature proves the impossibility of distributed consensus in asynchronous systems under faults [10]. Given a model of asynchronous communication, the authors show that consensus cannot be achieved when even one participant fails. The truth of the result, under the starting conditions of the analysis, is indisputable and relies on mathematical logic. Engineering, however (see below), can make use of this truth as necessary; for instance, an actual protocol or application that ensures it does not fall under the model of the impossibility result can have hopes of achieving distributed consensus.

In some cases, stepping back and looking at the behavior of large populations of human-made artifacts in the proper scope can yield to scientific study of a manner similar to that employed in physical sciences. Of particular importance in the networking community has been the identification of self-similarity in network traffic [15]. This work establishes that network packet traffic has self-similar characteristics, by performing a thorough statistical analysis of a very large set of Ethernet packet traces. Before this work was published, "truth" was that network traffic could be modeled as a Poisson process, deeply affecting every analysis of networks. For example, in the previous model, traffic aggregation was thought to help "smooth" bursts in traffic, which conflicted with practical observation; this work identified the reasons for this conflict.

Finally, a third example demonstrates truth at the boundaries of engineering, by identifying structure within artifacts imposed on the physical world. Work on the duality of operating systems structures [14] proposes a fundamental set of principles mapping between two competing system engineering disciplines, and demonstrates a fundamental "truth": no matter how you slice them, message-oriented systems and procedure-oriented systems are equivalent and can only be differentiated in terms of engineering convenience, not by their inherent strengths or weaknesses compared to each other. Instead of discovering a truth that was out there, this work looks at the effects of design choices to the physical world and identifies structure within them. In this case, the discovered structure is a duality.

2.2 Engineering

Whereas science seeks to uncover truth, regardless of where that truth resides or how it can be brought to bear on practice, engineering starts with the inevitability of practical relevance and goes backward to the principles that make practical utility achievable. In the example of distributed consensus above, science demanded that a universally true statement be made. However, where does this statement leave the systems researcher who still needs to deal with fault-tolerant consensus in practical systems? The answer lies in the engineering pursuit of achieving a solution that works for a particular problem and might not, necessarily, generalize; it lies as well in the analysis of such heuristics to understand when they are "good enough" for practical use. For example, Castro and Liskov [8] employ a type of weak synchrony to exit the asynchronous regime of [10] and, thereby, achieve fault-tolerant, distributed consensus for replicated services. A practical engineering decision was made (imposing some restrictions on how communication is conducted) to enable a solution for a real problem.

Unlike this example, in many cases, engineering systems research presents no new truths; it deals with solving a particular problem by synthesizing truths and solutions previously proposed. When the problem is shared by a large population, the utility of such a solution can outstrip significantly the utility of a new truth, at least in the short term. The Google File System, for instance [11], is a layer that supports the Internet searches of millions of users every day. Its authors admit that its design is not intended to be general or even applicable to any other storage problem. Yet, the broad relevance of the target application makes this engineering effort worthwhile and significant.

2.3 Art

In systems research, art has been a controversial evaluation dimension. Its manifestations as elegance, or, when stretched to its subjective extreme, beauty, can make complex ideas more palatable or more comprehensible. Elegance, as economy of expression in system abstractions, interfaces, and languages may help to sell the argument behind a complex idea [20], or bring order in an area where chaos reigned before. It it also a key contributor to the user experience for computer systems, affecting long-term ownership and maintenance costs: systems with elegance in their underlying designs are often easier to use and manage as well as being less prone to human error. On the other hand, elegance is sometimes parsimonious, economic but hard to comprehend [5]. Finally, often artful system design is its own goal.

The classic THE multiprogramming system [9] can be considered an example of elegant, simple, harmonious system design. Layering is pushed to the extreme, pro-

viding for a clean separation of concerns and a design that promotes composable verification of individual system components. Though an early version of a complex software system, this work exemplified the beauty and elegance of clean interfaces to enhance system understandability. In a more general way, BAN logic [7] introduced a simple, contained structure to reasoning about authentication protocols, shaping an entire field for years to come. On the side of engineering elegance, Tangler [19] is a collaborative storage system that balances function with participation incentives. To ensure his documents are protected against censorship a user must employ and remember a source of randomness found in another user's documents. Incentives for storing foreign content are balanced by a user's need to retrieve his own content.

As in all art, the beauty in elegant system research often lies in the (subjective) eye of the beholder. In the examples above, the THE system sacrificed efficiency for strict layering, BAN logic lacked an elegant path for adoption by error-prone humans, and Tangler was a niche application.

2.4 Discussion

One could argue that influential systems research scores high on multiple dimensions. An elegant and scientifically sound study is strictly better (more understandable, more extensible, etc.) than a sterile, unintuitive yet correct study. For example, Gummadi et al. [12] present a great example of engineering elegance, by abstracting away implementation-specific details of different distributed hash table algorithms and distilling a simple engineering rule for the selection of the geometry of overlays. On the other hand, elegance that is patently false is often the weapon of a demagogue. As we illustrate in the following section, when evaluating systems research, elegance cannot replace correctness.

3 Evaluation Criteria

Each of the dimensions introduced in the previous section requires its own set of evaluation criteria. In this section we develop these criteria, treating the dimensions as independent; work that spans multiple dimensions should be assessed on the aggregate criteria of all dimensions touched.

3.1 Science

The value of systems work that falls into the science category lies in its ability to expose new truths about the constructs, abstractions, architectures, algorithms, and implementation techniques that make up the core of systems research. On one hand, evaluation criteria must fathom the depth of the truths revealed, based on novelty, their ability to categorize and explain existing constructs or behaviors, and their generality and applicability to multiple platforms and environments.

Papers that build a comprehensive design space around a set of ad-hoc techniques (like peer-to-peer routing protocols), or those that provide a foundation for understanding the tradeoffs in the use of different constructs (like threads vs. events, VMMs vs. microkernels), are stronger science than those that simply provide engineering insight into the behavior of a particular system implementation.

On the other hand, it is important to evaluate the methodological rigor of systems science. The essence of science — the scientific method — involves the careful testing or mathematical proof of an explicitly-formed hypothesis. Evaluators of science should look for work that forms a clear hypothesis, that constructs reproducible experiments to shed light on that hypothesis while controlling other variables, and that includes the analysis needed to prove or disprove the hypothesis. In particular, careful measurements are essential to strong science work.

3.2 Engineering

The value of systems work that falls into the engineering category lies in its utility: the breadth of applicability of the engineering technique to important real-world contexts, and the power of the technique to solve important problems in those contexts. Engineering work that succeeds on the first criterion will define techniques that open up a broad space of new applications—such as the introduction of BPF [17], which enabled a large body of work on such varied topics as intrusion detection, worm filtering, and tunneling---or that addresses a persistent problem that appears in many contexts, such as caching. Work that succeeds on the second criterion will typically introduce a method for solving a problem that is more effective than any existing known solution—a "best of breed" technique. The best engineering work succeeds on both criteria, introducing powerful solutions to broadly-applicable problems. Work that only addresses one criterion must be examined carefully relative to its practical utility; for example, work that provides a powerful solution to a non-problem ("engineering for its own sake") does not represent high-value engineering research, although it might fall into the category of art.

Another key criterion for evaluating engineering work, especially in the form of a research paper, is the strength of its evaluation. Good engineering work includes detailed measurements that demonstrate the value of the work along both the power and applicability axes. The latter is key—a paper that claims broad applicability but only measures its technique on a series of microbenchmarks does not demonstrate its value as well as one that analyzes the technique's power across a variety of realistic environments.

3.3 Art

Evaluating systems research that falls into the art category is inherently a subjective business. The typical evaluation criteria for art include elegance, beauty, simplicity, and its ability to introduce new perspectives on existing, well-trodden areas. None of these are easily quantified except perhaps the last, and even that is subject to interpretation. Human factors studies (such as those commonly found in HCI research) are perhaps a first step toward collecting and correlating evaluations of aspects of art, such as usability or elegance, and perhaps should be seen more frequently in systems research work; however, they remain based on subjective assessments. Thus, since there will likely always be some disagreement about artistic value, "artful" systems research is best left to be evaluated by its consumers and the impartial view of history. A more practical approach, perhaps, is to use a panel of expert judges as is done in other artistic fields; the existing construct of a program committee fits well into this paradigm although it can suffer the same capriciousness that plagues judging in the arts.

4 A Prescription for More Rigorous Evaluation

While the systems research community has an excellent track record of producing high-quality, high-impact research in all three dimensions of science, engineering, and art, it has historically fallen short in evaluating that work with the rigor and discipline of other scientific and engineering communities. This weakness can be attributed to many factors, including the field's relative youth and its tight association with the fast-moving marketplace, but two stand out in particular: (1) a lack of solid methodology for scientific and engineering evaluation, and (2) a lack of recognition that some systems work is art and must be evaluated as such. As an impetus to remedy this situation, the following sections propose guidelines and research directions to help steer the community toward more rigorous and effective evaluation.

4.1 Science: Revive the Scientific Method

Science is defined by the scientific method, namely the identification of a hypothesis, reproducible collection of experimental data related to that hypothesis, and analysis of the data to evaluate the validity of the hypothesis. Systems research that falls along the science dimension must be evaluated with respect to how well it implements the scientific method.

For the researcher, that means several things. Most important is establishing a well-defined hypothesis. At one extreme, this could be a theorem to be proven; it could also be a claim about system behavior, for example that the same scalability is achievable with threaded architectures as with event-driven architectures. The hypothesis must then be followed up with a set of well-designed, reproducible experiments that illuminate the hypothesis and control for unrelated variables. When control is not possible, enough data must be collected to allow a statistical analysis of the effect of the uncontrolled variables. For example, in the threads-vs.-events hypothesis above, a good set of experiments will control the application, platform, workload, and quality of the implementations being evaluated; if the implementation quality could not be controlled, the experiments should collect data on multiple implementations.

Finally, good science-style systems research must include a sound analysis of the experimental data relative to the hypothesis. A key aspect missing from much systems research is the use of statistics and statistical tests to analyze experimental data---just compare the typical paper in the biological or physical sciences to the typical systems research paper to see the difference! Systems researchers should learn and use the toolbox of statistical tests available to them; systems papers should start reporting p-values to support claims that experimental data proves a hypothesis.

And those evaluating completed systems work—like program committees—should look for and insist on rigorous application of the scientific method, including well-defined hypotheses, reproducible experiments, and the kind of rigorous statistical analysis that we advocate. They should look for experiments and data that directly assess the hypothesis, not just that provide numbers—there are many hypotheses in systems research, particularly in the new focus areas of dependability and reliability, that are not proven by lists of performance figures. Since reproducibility is a key aspect of the scientific method, the community should also provide forums for publishing reproductions of key system results—perhaps as part of graduate student training or in special sessions at key conferences and workshops.

4.2 Engineering: Focus on Real-World Utility

As described in Section 3, the key criterion for evaluating engineering work is applicability. For the researcher, this means that good engineering systems work (and the papers that describe it) will include evaluations illustrating the work's utility in real-world situations. This is a challenge for much modern systems research, since our evaluation metrics and methodologies are primarily built around performance assessment, and the utility challenges faced in many real environments center on other aspects like dependability, maintainability, usability, predictability, and cost. Another challenge is that it is often impractical to evaluate research work directly in the context of real-world deployments or laboratory mock-ups of such systems, so surrogate environments,

such as those defined by application benchmarks, must be used instead.

Thus there are two critical research challenges that must be addressed before we can easily evaluate systems research as to real-world utility. The first involves creating the surrogate environments that researchers, particularly academic researchers, can use to recreate real-world problems and demonstrate the applicability of new engineered technology. Accomplishing this goal will require increased cooperation between industry and academia, and in particular finding ways to transfer the applications and technology behind real-world systems to the academic community. We believe this to be a priority for the continued success and relevance of the systems research community, and call on industry leaders to find solutions to the current stumbling blocks of intellectual property restrictions and licensing costs.

Another promising possibility for creating surrogate environments is to explore ways to do for core systems research what PlanetLab did for distributed and networked computing research: create a shared, commonly-available, realistic mock-up of the complex deployments found in real production environments. For example, a consortium of researchers (academic and industrial) could be assembled to build and operate a production-grade enterprise service (such as a supply-chain operation or an online multiplayer game), providing a test bed for evaluating systems research technologies in the resiliency, security, and maintainability spaces, while sharing the burden of constructing and operating the environment.

The second challenge is to significantly advance our ability to evaluate aspects of utility other than performance. This implies research into metrics, reproducible methodologies, and realistic workloads—benchmarks—for non-performance aspects of systems engineering. Initial work has begun on benchmarks for dependability and usability [6, 13], but much additional research is needed. This effort will require research along all three dimensions of systems research—art to figure out how to approach the problem, science to develop and test methodologies and metrics, and engineering to implement them as benchmarks—and its success will be evaluated by the improvements in rigor and applicability we can achieve in assessing the real-world utility of engineering-style systems research.

4.3 Art: Legitimize Artistic Research

Despite the inherent subjectivity in its evaluation, "artistic" systems research can have tremendous value, particularly in spurring the development of new subfields and in laying the foundation for future advances in science and engineering. But because this value is judged subjectively, artistic research cannot (and should not) be evaluated on the same scale as scientific- or engineering-style research—i.e., one should not demand quantitative results in a paper whose primary contribution is artistic. Today, most forums for publication and discussion receive a mix of artistic, scientific, and engineering submissions, biasing the selection away from the necessarily less-quantified artistic submissions. Instead, the systems community needs to create spaces where artistic research can be presented, examined, and subjectively judged against other artistic research. How this is best accomplished is an open question, since there is a definite risk of marginalizing art papers if handled incorrectly, but one possibility is to dedicate a certain fraction of the paper slots or sessions at the major conferences (or issues of the major journals) to artistic research, perhaps even with a separate reviewing process than the remainder of the conference or journal.

5 Related Work

Related work in this area is available mostly in the form of referee guidelines published by conference program committees and paper evaluation criteria presented in calls for papers. Guidelines for conference referees usually ask committee members to evaluate the degree of technical contribution, novelty, originality and importance to the community [1, 2]. A typical call for papers suggests that a good systems paper would have attacked a significant problem, demonstrated advancement beyond previous work, devised a clever solution and argued its practicality, and drawn appropriate conclusions [3, 4]. Their proposed criteria are overly general and may not fit all types of systems project equally well.

Patterson suggested that the principal criterion for evaluating research is its long-term impact on the technology [18]. While this is a reasonable criterion for a long-running project, it cannot be easily applied to new research, because it is difficult to envision the long-term impact that this research will produce.

Work by Levin and Redell, Ninth SOSP Committee co-chairmen, is perhaps the closest to our work, and is one of the first publications describing a systematic process of evaluating systems research [16]. Like us, they state that there exist different classes of research and that different criteria should be applied for different classes. They propose the following evaluation criteria: originality of ideas, availability of real implementations, importance of lessons learned, extent to which alternative design choices were explored and soundness of assumptions. They describe how to apply these criteria and emphasize which criteria are more appropriate for a particular type of research. In contrast, the contribution of our work is categorization of criteria along the dimensions of science, engineering, and art, as well as the description

of criteria for each dimension and suggestions on how to incorporate those in the evaluation of systems research.

6 Conclusions

Systems research is difficult to evaluate because of its multidimensional nature. In this paper we have identified three dimensions of systems research: science, engineering, and art. We mention several research papers in each of these three domains. For each of these domains we have outlined desirable characteristics to conduct and present research work. Because of the multidimensional nature of systems research, we argue for dimension-specific evaluation criteria. In this regard, we suggest a set of evaluation guidelines for the above mentioned three dimensions. We propose that scientific research works be evaluated by how strictly they adhere to the rigors of scientific methodology; that utility and applicability be the yardstick for engineering research works; and that, in the category of art, research works be judged by their elegance and simplicity. By guiding researchers to better conduct and present their work, and reviewers to evaluate publications with applicable criteria, we believe that this discussion may prove beneficial in improving the systems research landscape.

7 Acknowledgements

The material in this paper was initially developed during a breakout session at the Tenth Workshop on Hot Topics in Operating Systems (HotOS-X). We wish to recognize and thank the following participants as key contributors to the ideas and content of this paper: Pei Cao, Ira Cohen, Robert Grimm, Gernot Heiser, Sharon Perl, Ion Stoica, and Xiaoyun Zhu.

References

[1] 1999 PPoPP electronic referee's report form. `http://www.cs.utah.edu/~wilson/compilers/review-form1.txt`, 1999.

[2] PLDI 2001 referee's report form. `http://www.cs.utah.edu/~wilson/compilers/review-form2.txt`, 2001.

[3] USENIX 2002 call for papers. `http://www.usenix.org/events/usenix02/cfp/usenix02cfp.pdf`, 2002.

[4] OSDI 2004 call for papers. `http://www.usenix.org/events/osdi04/cfp/`, 2004.

[5] BERKHEIMER, G. D., ANDERSON, C. W., AND SPEES, S. T. Using conceptual change research to reason about curriculum. *Research Series No. 195, Michigan State University, Institute for Research on Teaching* (1989).

[6] BROWN, A. B., AND PATTERSON, D. A. Towards availability benchmarks: A case study of software RAID systems. In *USENIX '00: Proceedings of the USENIX Annual Technical Conference* (San Diego, California, USA, June 2000), USENIX Association.

[7] BURROWS, M., ABADI, M., AND NEEDHAM, R. A logic of authentication. *ACM Trans. Comput. Syst. 8*, 1 (1990), 18–36.

[8] CASTRO, M., AND LISKOV, B. Practical byzantine fault tolerance. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation* (New Orleans, Louisiana, United States, 1999), USENIX Association, pp. 173–186.

[9] DIJKSTRA, E. W. The structure of the "THE"-multiprogramming system. *Commun. ACM 11*, 5 (1968), 341–346.

[10] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *J. ACM 32*, 2 (1985), 374–382.

[11] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles* (Bolton Landing, NY, USA, 2003), ACM Press, pp. 29–43.

[12] GUMMADI, K., GUMMADI, R., GRIBBLE, S., RATNASAMY, S., SHENKER, S., AND STOICA, I. The impact of DHT routing geometry on resilience and proximity. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2003), ACM Press, pp. 381–394.

[13] KANOUN, K., MADIERA, H., AND ARLAT, J. A framework for dependability benchmarking. In *DSN '02: Proceedings of the Workshop on Dependability Benchmarking* (Washington, DC, United States, June 2002).

[14] LAUER, H. C., AND NEEDHAM, R. M. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev. 13*, 2 (1979), 3–19.

[15] LELAND, W. E., TAQQU, M. S., WILLINGER, W., AND WILSON, D. V. On the self-similar nature of ethernet traffic (extended version). *IEEE/ACM Trans. Netw. 2*, 1 (1994), 1–15.

[16] LEVIN, R., AND REDELL, D. D. An Evaluation of the Ninth SOSP Submissions or How (and How Not) to Write a Good Systems Paper. `http://www.usenix.org/events/samples/submit/advice.html`, 1983.

[17] MCCANNE, S., AND JACOBSON, V. The BSD packet filter: A new architecture for user-level packet filtering. In *USENIX '93: Proceedings of the USENIX Annual Technical Conference* (San Diego, California, USA, Jan. 1993), USENIX Association.

[18] PATTERSON, D. How to have a bad career in research/academia. `http://www.cs.berkeley.edu/~pattrsn/talks/nontech.html`.

[19] WALDMAN, M., AND MAZIERES, D. Tangler: a censorship-resistant publishing system based on document entanglements. In *CCS '01: Proceedings of the 8th ACM conference on Computer and Communications Security* (New York, NY, USA, 2001), ACM Press, pp. 126–135.

[20] WILLIAMS, R. Context, Content and Commodities: e-Learning Objects. In *Proceedings of the 2003 European Conference on eLearning* (Glasgow, UK, Nov. 2003), pp. 485–494.

Notes

[1] Under the term "systems research" we bundle any work that would come out of a "systems group" at a research university, including not only Operating Systems, but networking, distributed systems, theory about systems, etc. In short, we consider work that would conceivably appear in the proceedings of HotOS, OSDI, NSDI, SOSP, etc.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of system administrators, developers, programmers, and engineers working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering technical excellence and innovation
- encouraging computing outreach in the community at large
- providing a neutral forum for the discussion of critical issues

Membership Benefits

- Free subscription to *;login:*, the Association's magazine, both in print and online
- Online access to all Conference Proceedings from 1993 to the present
- Access to the USENIX Jobs Board: Perfect for those who are looking for work or are looking to hire from the talented pool of USENIX members
- The right to vote in USENIX Association elections
- Discounts on technical sessions registration fees for all USENIX-sponsored and co-sponsored events
- Discounts on purchasing printed Proceedings, CD-ROMs, and other Association publications
- Discounts on industry-related publications; see http://www.usenix.org/membership/specialdisc.html

For more information about membership, conferences, or publications, see http://www.usenix.org.

USENIX Thanks Its Supporting Members

USENIX Supporting Members

Addison-Wesley/Prentice Hall PTR • AMD • Asian Development Bank
Cambridge Computer Services, Inc. • Delmar Learning • Electronic Frontier Foundation
Eli Research • GroundWork Open Source Solutions • Hewlett-Packard • IBM • Intel
Interhack • The Measurement Factory • Microsoft Research • NetApp • Oracle • OSDL
Perfect Order • Raytheon • Ripe NCC • Splunk • Sun Microsystems, Inc. • Taos
Tellme Networks • UUNET Technologies, Inc.

ISBN 1-931971-36-6